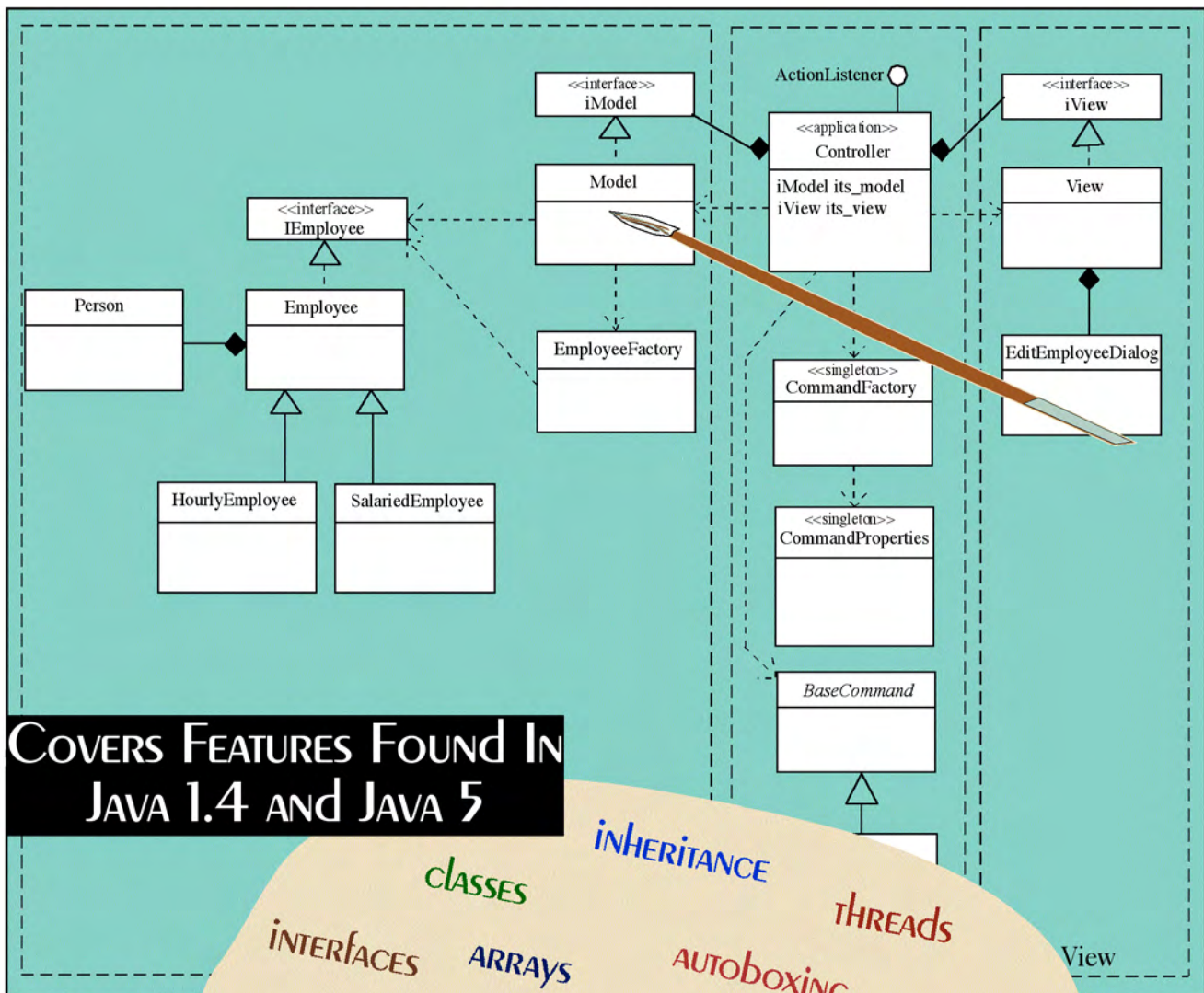
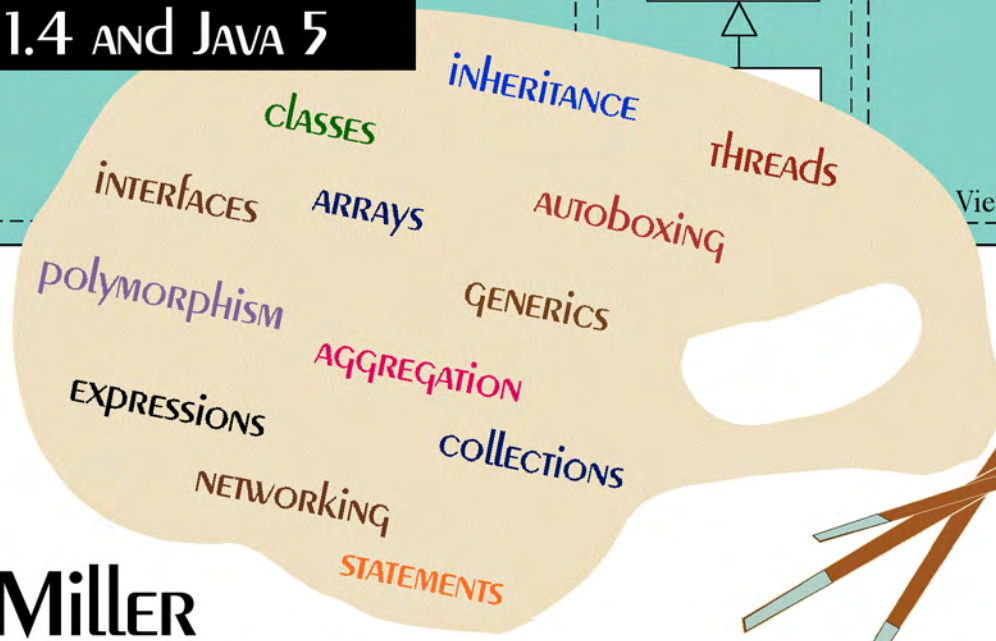


JAVA FOR ARTISTS

THE ART, PHILOSOPHY, AND SCIENCE OF OBJECT-ORIENTED PROGRAMMING



COVERS FEATURES FOUND IN
JAVA 1.4 AND JAVA 5



Rick Miller
Raffi KASPARIAN

Supplemental CD Included! UNLEASH THE CREATIVE GENIUS IN YOU!

JAVA FOR ARTISTS

JAVA FOR ARTISTS

THE ART, PHILOSOPHY, AND SCIENCE OF OBJECT-ORIENTED PROGRAMMING

RICK MILLER
RAFFI KASPARIAN

PULP FREE PRESS
FALLS CHURCH, VIRGINIA

Pulp FREE PRESS, Falls Church, Virginia 22042

www.pulpfreepress.com

info@pulpfreepress.com

©2006 Richard Warren Miller, Raffi Kasparian, & Pulp Free Press — All Rights Reserved

No part of this book may be reproduced in any form without prior written permission from the publisher.

First edition published 2006

16 14 12 10 08 06

10 9 8 7 6 5 4 3 2

Pulp Free Press offers discounts on this book when ordered in bulk quantities. For more information regarding sales contact sales@pulpfreepress.com.

The eBook/PDF edition of this book may be licensed for bulk distribution. For whole or partial content licensing information contact licensing@pulpfreepress.com.

Publisher Cataloging-in-Publication Data: *Prepared by Pulp Free Press*

Miller, Rick, 1961 - , Kasparian, Raffi, 1962 -
Java For Artists: The Art, Philosophy, and Science of Object-Oriented
Programming/Rick Miller and Raffi Kasparian.

p. cm.

Includes bibliographical references and index.

ISBN-13: 978-1-9325-0405-7 ISBN-10: 1-932504-05-2 (pbk)

1. Java (computer program language) I. Raffi Kasparian II. Title.

2. Object-Oriented Programming (Computer Science)

QA76.73.J38M555 2006

005.13'3--dc21

Library of Congress Control Number: 2005902971

CIP

The source code provided in this book is intended for instructional purposes only. Although every possible measure was made by the authors to ensure the programming examples contain error-free source code, no warranty is offered, expressed, or implied regarding the quality of the code.

All product names mentioned in this book are trademark names of their respective owners.

Java For Artists was meticulously crafted on a Macintosh PowerMac G4 using Adobe FrameMaker, Adobe Illustrator, Macromedia Freehand, Adobe Photoshop, Adobe Acrobat, ObjectPlant, Microsoft Word, Maple, and VirtualPC. Photos were taken with a Nikon F3HP, a Nikon FM, and a Contax T3. Java source code examples were prepared using Apple's OS X developer tools, vi, TextPad, Eclipse, and Sun's Java 2 Standard Edition (J2SE versions 1.4 and 1.5) command-line tool suite.

ISBN-10: 1-932504-03-6 First eBook/PDF Edition

ISBN-10: 1-932504-04-4 First CD ROM Edition

ISBN-10: 1-932504-05-2 First Paperback Edition

Again — to Coralie and Kyle for their eternal patience.

Rick

To my lovely wife Joycelyn for her patience and support and especially for introducing me to Java way back when it was version 1.0.

Raffi

DETAILED CONTENTS

PREFACE

WELCOME – AND THANK YOU!	xlvi
TARGET AUDIENCE	xlvi
APPROACH(ES)	xlvi
ORGANIZATION	xlvi
<i>PART I: THE JAVA STUDENT SURVIVAL GUIDE</i>	<i>xlvi</i>
CHAPTER 1 - AN APPROACH TO THE ART OF PROGRAMMING	<i>xlvi</i>
CHAPTER 2 - SMALL VICTORIES.....	<i>xlvi</i>
CHAPTER 3 - PROJECT WALKTHROUGH: A COMPLETE EXAMPLE	<i>xlvii</i>
CHAPTER 4 - COMPUTERS, PROGRAMS, AND ALGORITHMS	<i>xlvii</i>
<i>PART II: LANGUAGE FUNDAMENTALS</i>	<i>xlvii</i>
CHAPTER 5 - OVERVIEW OF THE JAVA API.....	<i>xlvii</i>
CHAPTER 6 - SIMPLE JAVA PROGRAMS: USING PRIMITIVE AND REFERENCE DATA TYPES.....	<i>xlvii</i>
CHAPTER 7 - CONTROLLING THE FLOW OF PROGRAM EXECUTION	<i>xlvii</i>
CHAPTER 8 - ARRAYS.....	<i>xlvii</i>
CHAPTER 9 - TOWARD PROBLEM ABSTRACTION: CREATING NEW DATA TYPES	<i>xlviii</i>
CHAPTER 10 - COMPOSITIONAL DESIGN	<i>xlviii</i>
CHAPTER 11 - EXTENDING CLASS BEHAVIOR THROUGH INHERITANCE	<i>xlviii</i>
<i>PART III: GRAPHICAL USER INTERFACE PROGRAMMING</i>	<i>xlviii</i>
CHAPTER 12 - JAVA SWING API OVERVIEW.....	<i>xlviii</i>
CHAPTER 13 - HANDLING GUI EVENTS.....	<i>xlviii</i>
CHAPTER 14 - AN ADVANCED GUI PROJECT	<i>xlviii</i>
<i>PART IV: INTERMEDIATE CONCEPTS</i>	<i>xl ix</i>
CHAPTER 15 - EXCEPTIONS.....	<i>xl ix</i>
CHAPTER 16 - THREADS.....	<i>xl ix</i>
CHAPTER 17 - COLLECTIONS.....	<i>xl ix</i>
CHAPTER 18 - FILE I/O.....	<i>xl ix</i>
<i>PART V: NETWORK PROGRAMMING</i>	<i>xl ix</i>
CHAPTER 19 - INTRODUCTION TO NETWORKING AND DISTRIBUTED APPLICATIONS	<i>l</i>
CHAPTER 20 - CLIENT-SERVER APPLICATIONS.....	<i>l</i>
CHAPTER 21 - APPLETS AND JDBC.....	<i>l</i>
<i>PART VI: OBJECT-ORIENTED PROGRAMMING</i>	<i>l</i>
CHAPTER 22 - INHERITANCE, COMPOSITION, INTERFACES, AND POLYMORPHISM.....	<i>l</i>
CHAPTER 23 - WELL-BEHAVED OBJECTS.....	<i>li</i>
CHAPTER 24 - THREE DESIGN PRINCIPLES.....	<i>li</i>
CHAPTER 25 - HELPFUL DESIGN PATTERNS	<i>li</i>
Pedagogy	li
<i>LEARNING OBJECTIVES</i>	<i>li</i>
<i>INTRODUCTION</i>	<i>li</i>
<i>CONTENT</i>	<i>li</i>
<i>Quick Reviews</i>	<i>li</i>
<i>SUMMARY</i>	<i>lii</i>
<i>Skill-Building Exercises</i>	<i>lii</i>
<i>SUGGESTED PROJECTS</i>	<i>lii</i>
<i>SELF-TEST QUESTIONS</i>	<i>lii</i>
<i>REFERENCES</i>	<i>lii</i>
<i>NOTES</i>	<i>lii</i>
Typographical FORMATS	lii
<i>This Is An Example Of A First-Level Subheading</i>	<i>lii</i>

<i>This Is An Example Of A Second-Level Subheading</i>	lii
SOURCE CODE FORMATTING	liii
CD-ROM	liii
SUPPORTSITE™ WEBSITE	liii
PROBLEM REPORTING	liii
ABOUT THE AUTHORS	liii
ACKNOWLEDGMENTS	liv

PART I: THE JAVA STUDENT SURVIVAL GUIDE

I AN APPROACH TO THE ART OF PROGRAMMING

INTRODUCTION	4
<i>The Difficulties You Will Encounter Learning Java</i>	4
REQUIRED SKILLS	4
THE PLANETS WILL COME INTO ALIGNMENT.....	4
How This Chapter Will Help You	5
PROJECT MANAGEMENT	5
THREE SOFTWARE DEVELOPMENT ROLES	5
ANALYST	5
ARCHITECT.....	5
PROGRAMMER	6
A PROJECT-APPROACH STRATEGY	6
YOU HAVE BEEN HANDED A PROJECT – NOW WHAT?.....	6
STRATEGY AREAS OF CONCERN.....	6
THINK ABSTRACTLY.....	8
THE STRATEGY IN A NUTSHELL	8
APPLICABILITY TO THE REAL WORLD	8
THE ART OF PROGRAMMING	8
DON'T START AT THE COMPUTER	9
INSPIRATION STRIKES AT THE WEIRDEST TIME	9
OWN YOUR OWN COMPUTER	9
YOU EITHER HAVE TIME AND NO MONEY, OR MONEY AND NO TIME	9
THE FAMILY COMPUTER IS NOT GOING TO CUT IT!	9
SET THE MOOD	10
LOCATION, LOCATION, LOCATION.....	10
CONCEPT OF THE FLOW	10
THE STAGES OF FLOW.....	10
BE EXTREME	11
THE PROGRAMMING CYCLE.....	11
THE PROGRAMMING CYCLE SUMMARIZED.....	12
A HELPFUL TRICK: STUBBING	12
FIX THE FIRST COMPILER ERROR FIRST	12
MANAGING PROJECT COMPLEXITY	12
CONCEPTUAL COMPLEXITY	12
MANAGING CONCEPTUAL COMPLEXITY.....	13
PHYSICAL COMPLEXITY	13
THE RELATIONSHIP BETWEEN PHYSICAL AND CONCEPTUAL COMPLEXITY.....	14
MANAGING PHYSICAL COMPLEXITY.....	14
MAXIMIZE COHESION – MINIMIZE COUPLING	14
JAVA SOURCE FILE STRUCTURE	14
SAMPLE CLASS IN ACTION	15
GENERAL RULES FOR CREATING JAVA SOURCE FILES	16

<i>Rule-of-Thumb: ONE CLASS PER FILE</i>	17
<i>Avoid ANONYMOUS, NESTED, AND INNER CLASSES</i>	17
<i>CREATE A SEPARATE MAIN APPLICATION FILE</i>	17
PACKAGES	17
COMMENTING	18
<i>SINGLE-LINE COMMENTS</i>	18
<i>MULTI-LINE COMMENTS</i>	18
<i>JAVADOC COMMENTS</i>	19
<i>GENERATING JAVADOC EXAMPLE OUTPUT</i>	19
IDENTIFIER NAMING - WRITING SELF-COMMENTING CODE	19
<i>BENEFITS OF SELF-COMMENTING CODE</i>	20
<i>CODING CONVENTION</i>	20
<i>CLASS NAMES</i>	21
<i>CONSTANT NAMES</i>	21
<i>VARIABLE NAMES</i>	21
<i>METHOD NAMES</i>	22
SUMMARY	22
SKILL-BUILDING EXERCISES	22
SUGGESTED PROJECTS	24
SELF-TEST QUESTIONS	24
REFERENCES	25
NOTES	25

2 SMALL VICTORIES: CREATING JAVA PROJECTS

INTRODUCTION	28
JAVA PROJECT CREATION PROCESS	28
<i>SOURCE FILE CREATION STAGE</i>	29
<i>SOURCE FILE COMPILATION STAGE</i>	29
<i>WHAT IF A SOURCE FILE CONTAINS ERRORS?</i>	29
<i>MAIN APPLICATION EXECUTION STAGE</i>	29
<i>AUTOMATING THE JAVA PROJECT CREATION PROCESS</i>	30
INTEGRATED DEVELOPMENT ENVIRONMENTS	30
<i>TEXTPAD™ - J2SDK COMBINATION</i>	30
<i>SUN'S NETBEANS™</i>	30
<i>BORLAND® JBUILDER®</i>	30
<i>ECLIPSE®</i>	30
CREATING JAVA PROJECTS USING MICROSOFT® WINDOWS® 2000/XP	31
<i>SUN'S JAVA 2 STANDARD EDITION SOFTWARE DEVELOPMENT KIT (J2SDK)</i>	31
<i>STEP 1: DOWNLOAD AND INSTALL THE J2SDK</i>	31
<i>STEPS 2 & 3: SETTING MICROSOFT WINDOWS ENVIRONMENT VARIABLES</i>	31
<i>ANOTHER HELPFUL HINT: DISPLAY FILES SUFFIXES AND FULL PATH NAMES</i>	36
<i>STEPS 4 & 5: SELECT A TEXT EDITOR AND CREATE THE SOURCE FILES</i>	36
<i>STEP 6: COMPILING THE SOURCE FILES</i>	38
<i>STEP 7: EXECUTING THE MAIN APPLICATION CLASS</i>	38
<i>BORLAND'S JBUILDER</i>	39
<i>STEPS TO CREATING JAVA PROJECTS IN JBUILDER</i>	39
<i>QUICK REVIEW</i>	45
CREATING JAVA PROJECTS USING LINUX™	45
<i>A BRIEF NOTE ON THE MANY FLAVORS OF LINUX</i>	46
<i>OBTAINING AND INSTALLING SUN'S J2SDK FOR LINUX</i>	46
<i>SELECTING AN INSTALLATION LOCATION</i>	47
<i>SETTING THE PATH ENVIRONMENT VARIABLE (BASH SHELL)</i>	48
<i>SETTING THE CLASSPATH ENVIRONMENT VARIABLE (BASH)</i>	49
<i>CREATING, COMPILING, AND RUNNING JAVA PROGRAMS</i>	49
<i>CREATING JAVA SOURCE FILES</i>	49

<i>Compiling JAVA SOURCE Files</i>	50
<i>Running The Application</i>	50
<i>Quick Review</i>	51
CREATING JAVA PROJECTS USING MACINTOSH OS X DEVELOPER TOOLS	51
<i>CREATING A JAVA PROJECT With XCODE</i>	51
<i>Quick Review</i>	52
CREATING AND RUNNING EXECUTABLE JAR FILES	53
<i>STEPS TO CREATE AN EXECUTABLE JAR FILE</i>	53
<i>STEP 1: CREATE YOUR JAVA PROGRAM</i>	53
<i>STEP 2: CREATE MANIFEST FILE</i>	54
<i>STEP 3: USE THE jar COMMAND</i>	54
<i>EXECUTING THE jar FILE</i>	54
<i>Quick Review</i>	54
SUMMARY	55
Skill-Building EXERCISES	55
SUGGESTED PROJECTS	56
SELF-TEST QUESTIONS	56
REFERENCES	57
NOTES	57

3 PROJECT WALKTHROUGH: A COMPLETE EXAMPLE

INTRODUCTION	60
THE PROJECT-APPROACH STRATEGY SUMMARIZED	60
DEVELOPMENT CYCLE	61
PROJECT SPECIFICATION	62
<i>ANALYZING THE PROJECT SPECIFICATION</i>	63
<i>APPLICATION REQUIREMENTS STRATEGY AREA</i>	63
<i>PROBLEM DOMAIN STRATEGY AREA</i>	64
<i>LANGUAGE FEATURES STRATEGY AREA</i>	66
<i>DESIGN STRATEGY AREA</i>	68
DEVELOPMENT CYCLE FIRST ITERATION	69
<i>PLAN (FIRST ITERATION)</i>	69
<i>CODE (FIRST ITERATION)</i>	70
<i>TEST (FIRST ITERATION)</i>	70
<i>INTEGRATE/TEST (FIRST ITERATION)</i>	70
DEVELOPMENT CYCLE SECOND ITERATION	70
<i>PLAN (SECOND ITERATION)</i>	71
<i>CODE (SECOND ITERATION)</i>	71
<i>TEST (SECOND ITERATION)</i>	71
<i>INTEGRATE/TEST (SECOND ITERATION)</i>	72
DEVELOPMENT CYCLE THIRD ITERATION	72
<i>PLAN (THIRD ITERATION)</i>	72
<i>CODE (THIRD ITERATION)</i>	73
<i>TEST (THIRD ITERATION)</i>	75
<i>INTEGRATE/TEST (THIRD ITERATION)</i>	75
DEVELOPMENT CYCLE FOURTH ITERATION	75
<i>PLAN (FOURTH ITERATION)</i>	75
<i>IMPLEMENTING STATE TRANSITION DIAGRAMS</i>	76
<i>IMPLEMENTING THE printFloor() METHOD</i>	77
<i>CODE (FOURTH ITERATION)</i>	77
<i>TEST (FOURTH ITERATION)</i>	79
<i>INTEGRATE/TEST (FOURTH ITERATION)</i>	79
DEVELOPMENT CYCLE FIFTH ITERATION	79
<i>PLAN (FIFTH ITERATION)</i>	80
<i>CODE (FIFTH ITERATION)</i>	81

<i>TEST (Fifth ITERATION)</i>	82
<i>INTEGRATE/TEST (Fifth ITERATION)</i>	82
SOME FINAL CONSIDERATIONS	82
COMPLETE ROBOTRAT.JAVA SOURCE CODE LISTING	84
SUMMARY	88
SKILL-BUILDING EXERCISES	88
SUGGESTED PROJECTS	88
SELF-TEST QUESTIONS	88
REFERENCES	89
NOTES	89

4 COMPUTERS, PROGRAMS, AND ALGORITHMS

INTRODUCTION	92
WHAT IS A COMPUTER?	92
<i>COMPUTER VS. COMPUTER SYSTEM</i>	92
<i>COMPUTER SYSTEM</i>	92
<i>PROCESSOR</i>	94
<i>THREE ASPECTS OF PROCESSOR ARCHITECTURE</i>	95
<i>FEATURE SET</i>	95
<i>FEATURE SET IMPLEMENTATION</i>	95
<i>FEATURE SET ACCESSIBILITY</i>	95
MEMORY ORGANIZATION	95
<i>MEMORY BASICS</i>	96
<i>MEMORY HIERARCHY</i>	96
<i>Bits, Bytes, Words</i>	96
<i>ALIGNMENT AND ADDRESSABILITY</i>	97
WHAT IS A PROGRAM?	98
<i>TWO VIEWS OF A PROGRAM</i>	98
<i>THE HUMAN PERSPECTIVE</i>	98
<i>THE COMPUTER PERSPECTIVE</i>	98
THE PROCESSING CYCLE	98
<i>FETCH</i>	99
<i>DECODE</i>	99
<i>EXECUTE</i>	99
<i>STORE</i>	99
<i>Why A PROGRAM CRASHES</i>	99
ALGORITHMS	99
<i>Good vs. Bad Algorithms</i>	100
<i>DON'T REINVENT THE WHEEL!</i>	102
THE JAVA HOTSPOT™ VIRTUAL MACHINE	102
<i>OBTAINING THE JAVA HOTSPOT™ VIRTUAL MACHINE</i>	102
<i>CLIENT & SERVER VIRTUAL MACHINES</i>	102
<i>CLASSIC VM vs. JIT vs. HOTSPOT™</i>	103
<i>JAVA HOTSPOT™ VIRTUAL MACHINE ARCHITECTURE</i>	103
SERVICES PROVIDED BY THE JAVA HOTSPOT™ VIRTUAL MACHINE AND THE JRE	103
SUMMARY	104
SKILL-BUILDING EXERCISES	104
SUGGESTED PROJECTS	104
SELF-TEST QUESTIONS	105
REFERENCES	105
NOTES	105

PART II: LANGUAGE FUNDAMENTALS

5 OVERVIEW OF THE JAVA PLATFORM API

INTRODUCTION	110
JAVA PLATFORM API PACKAGES	110
<i>Obtaining Detailed Java Platform API Information</i>	<i>111</i>
NAVIGATING A CLASS INHERITANCE HIERARCHY	112
<i>Deprecated Methods</i>	<i>115</i>
JAVA PLATFORM PACKAGES USED IN THIS BOOK	116
SUMMARY	117
SKILL-BUILDING EXERCISES	117
SUGGESTED PROJECTS	118
SELF-TEST QUESTIONS	118
REFERENCES	118
NOTES	119

6 Simple Java Programs: Using Primitive And Reference Data Types

INTRODUCTION	122
TERMINOLOGY	122
DEFINITION OF A JAVA PROGRAM	123
<i>Application Objects</i>	<i>123</i>
<i>Talking About Applications</i>	<i>123</i>
<i>Applet Objects</i>	<i>123</i>
CREATING SIMPLE JAVA PROGRAMS (APPLICATIONS)	123
<i>Structure Of A Java Application</i>	<i>124</i>
<i>Compiling And Executing SimpleApplication</i>	<i>125</i>
<i>Quick Review</i>	<i>125</i>
<i>Building Bigger Applications</i>	<i>126</i>
IDENTIFIERS AND IDENTIFIER NAMING RULES	126
<i>Well-Named Identifiers Will Simplify Your Life</i>	<i>127</i>
JAVA RESERVED KEYWORDS	127
JAVA TYPE CATEGORIES	129
<i>Primitive Data Types</i>	<i>129</i>
<i>Reference Data Types</i>	<i>130</i>
<i>Array Types</i>	<i>130</i>
WORKING WITH PRIMITIVE TYPES	130
<i>Application Class Structure</i>	<i>130</i>
<i>Definition Of The Term "Variable"</i>	<i>131</i>
<i>Declaring And Using Variables In The main() Method</i>	<i>131</i>
<i>Definition Of The Term "Constant"</i>	<i>132</i>
<i>Declaring And Using Constants In The main() Method</i>	<i>132</i>
<i>Getting Simple Input Into Your Program</i>	<i>133</i>
<i>Using The main() Method's String Array To Accept Input At Program Startup</i>	<i>133</i>
WORKING WITH REFERENCE TYPES	134
<i>Definition Of A Reference Variable</i>	<i>135</i>
<i>Definition Of An Object</i>	<i>135</i>
<i>Creating Objects with the new Operator</i>	<i>135</i>
<i>Garbage Collection</i>	<i>136</i>
ACCESSING CLASS MEMBERS FROM THE main() METHOD	136
<i>The main() Method Is A Static Method</i>	<i>137</i>
<i>Definition Of The Term Class Variable/Constant</i>	<i>137</i>

<i>Definition Of The Term Instance Variable/Constant</i>	137
<i>An Example</i>	137
STATEMENTS And Expressions	138
OPERATORS	139
<i>How To Read The Operator Table</i>	139
<i>Use Parentheses To Explicitly Force Precedence</i>	141
<i>Java Operators In Detail</i>	141
<i>Arithmetic Operators +, -, *, /</i>	141
<i>String Concatenation Operator +</i>	142
<i>Modulus (Integer Remainder) Operator %</i>	143
<i>Greater-Than/Less-Than Operators >, >=, <, <=</i>	143
<i>Equality And Inequality Operators ==, !=</i>	144
<i>Conditional AND and OR Operators &&, </i>	144
<i>Boolean AND, OR, XOR, and NOT Operators &, , ^, !</i>	145
<i>Ternary Conditional Operator ?:</i>	145
<i>Left Shift and Right Shift Operators <<, >>, >>></i>	146
<i>instanceof Operator</i>	147
<i>Unary Prefix And Postfix Increment And Decrement Operators ++, -</i>	147
<i>Member Access Operator</i>	147
<i>Bitwise AND, OR, XOR, and Complement Operators &, , ^, ~</i>	148
<i>Combination Assignment Operators +=, -=, *=, /=, <<=, >>=, >>>=, &=, =, ^=</i>	148
JAVA PRIMITIVE Type Wrapper Classes	149
SUMMARY	150
Skill-Building Exercises	150
SUGGESTED PROJECTS	151
Self-Test Questions	152
REFERENCES	152
NOTES	153

7 CONTROLLING THE FLOW OF PROGRAM EXECUTION

INTRODUCTION	156
SELECTION STATEMENTS	156
<i>if STATEMENT</i>	156
<i>Executing Code Blocks in if STATEMENTS</i>	157
<i>Executing Consecutive if STATEMENTS</i>	157
<i>if/else STATEMENT</i>	158
<i>Chained if/else STATEMENTS</i>	159
<i>switch STATEMENT</i>	159
<i>Nested Switch STATEMENT</i>	161
<i>Quick Review</i>	162
ITERATION STATEMENTS	162
<i>while STATEMENT</i>	162
<i>Personality of the while STATEMENT</i>	163
<i>do/while STATEMENT</i>	164
<i>Personality of the do/while STATEMENT</i>	164
<i>FOR STATEMENT</i>	165
<i>How The for STATEMENT is Related to the While STATEMENT</i>	165
<i>Personality of the for STATEMENT</i>	165
<i>Nesting Iteration STATEMENTS</i>	166
<i>Mixing Selection & Iteration STATEMENTS: A Powerful Combination</i>	166
<i>Quick Review</i>	168
BREAK AND CONTINUE	168
<i>Unlabeled break</i>	168
<i>Labeled break</i>	169
<i>Unlabeled CONTINUE</i>	170

<i>Labeled CONTINUE</i>	170
<i>Quick Review</i>	171
SELECTION AND ITERATION STATEMENT SELECTION TABLE	171
SUMMARY	173
SKILL-BUILDING EXERCISES	173
SUGGESTED PROJECTS	175
SELF-TEST QUESTIONS	176
REFERENCES	177
NOTES	177

8 ARRAYS

INTRODUCTION	180
WHAT IS AN ARRAY?	180
<i>Specifying Array Types</i>	181
<i>Quick Review</i>	182
FUNCTIONALITY PROVIDED BY JAVA ARRAY TYPES	182
<i>Array-Type Inheritance Hierarchy</i>	182
<i>The java.lang.reflect.Array Class</i>	182
<i>Special Properties Of Java Arrays</i>	183
<i>Quick Review</i>	183
CREATING AND USING SINGLE-DIMENSIONAL ARRAYS	183
<i>Arrays Of Primitive Types</i>	183
<i>How Primitive Type Array Objects Are Arranged In Memory</i>	184
<i>Calling Object And Class Methods On Array References</i>	185
<i>Creating Single-Dimensional Arrays Using Array Literal Values</i>	186
<i>Differences Between Arrays Of Primitive Types And Arrays Of Reference Types</i>	187
<i>Single-dimensional Arrays In Action</i>	187
<i>Message Array</i>	188
<i>Calculating Averages</i>	191
<i>Histogram: Letter Frequency Counter</i>	192
<i>Quick Review</i>	193
CREATING AND USING MULTIDIMENSIONAL ARRAYS	194
<i>Multidimensional Array Declaration Syntax</i>	194
<i>Memory Representation Of A Two Dimensional Array</i>	196
<i>Creating Multidimensional Arrays Using Array Literals</i>	196
<i>Ragged Arrays</i>	197
<i>Multidimensional Arrays In Action</i>	198
<i>Weighted Grade Tool</i>	198
<i>Quick Review</i>	199
THE MAIN() METHOD'S STRING ARRAY	200
<i>Purpose And Use Of The main() Method's String Array</i>	200
MANIPULATING ARRAYS WITH THE java.util.ARRAYS CLASS	201
JAVA API CLASSES USED IN THIS CHAPTER	202
SUMMARY	203
SKILL-BUILDING EXERCISES	203
SUGGESTED PROJECTS	204
SELF-TEST QUESTIONS	206
REFERENCES	207
NOTES	207

9 TOWARD PROBLEM ABSTRACTION: CREATING NEW DATA TYPES

INTRODUCTION	210
ABSTRACTION: Amplify THE ESSENTIAL – ELIMINATE THE IRRELEVANT	210

<i>Abstraction Is The Art Of Programming</i>	210
<i>Where Problem Abstraction Fits Into The Development Cycle</i>	210
<i>Creating Your Own Data Types</i>	211
<i>Case-Study Project: Write A People Manager Program</i>	211
<i>Quick Review</i>	213
The UML Class Diagram	213
<i>Quick Review</i>	214
Overview Of The Java Class Construct	214
<i>Four Categories Of Class Members</i>	214
<i>Static Or Class-Wide Fields</i>	214
<i>Non-Static Or Instance Fields</i>	215
<i>Static Or Class-Wide Methods</i>	215
<i>Non-Static Or Instance Methods</i>	215
<i>Access Modifiers</i>	215
<i>public</i>	215
<i>private</i>	215
<i>protected</i>	216
<i>package</i>	216
<i>The Concepts Of Horizontal Access, Interface, And Encapsulation</i>	216
<i>Quick Review</i>	217
Methods	217
<i>Method Naming – Use Action Words That Indicate The Method’s Purpose</i>	217
<i>Maximize Method Cohesion</i>	217
<i>Structure Of A Method Definition</i>	217
<i>Method Modifiers (optional)</i>	218
<i>Return Type Or void (optional)</i>	219
<i>Method Name (Mandatory)</i>	219
<i>Parameter List (optional)</i>	219
<i>Method Body (optional for abstract or native methods)</i>	219
<i>Example Method Definitions</i>	219
<i>Method Signatures</i>	220
<i>Overloading Methods</i>	220
<i>Constructor Methods</i>	220
<i>Quick Review</i>	220
Building And Testing The Person Class	221
<i>Start By Creating The Source File And Class Definition Shell</i>	221
<i>Defining Person Instance Fields</i>	221
<i>Defining Person Instance Methods</i>	221
<i>The Constructor Method</i>	222
<i>Adding A Few Accessor Methods</i>	222
<i>Testing The Person Class</i>	223
<i>Use The PeopleManagerApplication Class As A Test Driver</i>	223
<i>Adding Features To The Person Class – Calculating Age</i>	224
<i>Adding Features To The Person Class – Convenience Methods</i>	225
<i>Adding Features To The Person Class – Finishing Touches</i>	226
<i>Quick Review</i>	227
Building And Testing The PeopleManager Class	228
<i>Defining The PeopleManager Class Shell</i>	228
<i>Defining PeopleManager Fields</i>	228
<i>Defining PeopleManager Constructor Methods</i>	228
<i>Defining Additional PeopleManager Methods</i>	229
<i>Testing The PeopleManager Class</i>	230
<i>Adding Features To The PeopleManager Class</i>	230
<i>Quick Review</i>	232
More About Methods	232
<i>How Arguments Are Passed To Methods</i>	232
<i>Local Variable Scoping</i>	233
<i>Anywhere An Object Of <type> Is Required, A Method That Returns <type> Can Be Used</i>	233

<i>Quick Review</i>	234
Special Topics	234
<i>Static Initializers</i>	234
<i>Data Encapsulation - The Naked Truth</i>	235
<i>Quick Review</i>	236
SUMMARY	236
Skill-Building Exercises	237
SUGGESTED PROJECTS	238
SELF-TEST QUESTIONS	239
REFERENCES	241
NOTES	241

10 Compositional Design

INTRODUCTION	244
MANAGING CONCEPTUAL AND PHYSICAL COMPLEXITY	244
<i>Compiling Multiple Source Files Simultaneously with javac</i>	244
<i>Quick Review</i>	245
DEPENDENCY VS. ASSOCIATION	245
AGGREGATION	245
<i>Simple vs. Composite Aggregation</i>	246
<i>The Relationship Between Aggregation and Object Lifetime</i>	246
<i>Quick Review</i>	246
EXPRESSING AGGREGATION IN A UML CLASS DIAGRAM	246
<i>Simple Aggregation</i>	246
<i>Composite Aggregation</i>	247
AGGREGATION EXAMPLE CODE	247
<i>Simple Aggregation Example</i>	248
<i>Composite Aggregation Example</i>	249
<i>Quick Review</i>	250
SEQUENCE DIAGRAMS	250
<i>Quick Review</i>	251
THE AIRCRAFT ENGINE SIMULATION: AN EXTENDED EXAMPLE	251
<i>The Purpose of the Engine Class</i>	252
<i>Engine Class Attributes and Methods</i>	252
<i>PartStatus - A Typesafe Enumeration Pattern</i>	254
<i>Aircraft Engine Simulation Sequence Diagrams</i>	254
RUNNING THE AIRCRAFT ENGINE SIMULATION PROGRAM	256
<i>Quick Review</i>	256
COMPLETE AIRCRAFT ENGINE SIMULATION CODE LISTING	257
SUMMARY	260
Skill-Building Exercises	260
SUGGESTED PROJECTS	261
SELF-TEST QUESTIONS	262
REFERENCES	263
NOTES	263

11 Extending Class Behavior Through Inheritance

INTRODUCTION	266
THREE PURPOSES OF INHERITANCE	266
<i>Implementing The "is A" Relationship</i>	267
<i>The Relationship Between The Terms Type, Interface, and Class</i>	267
<i>Meaning Of The Term Interface</i>	267

<i>MEANING OF THE TERM CLASS</i>	267
<i>Quick Review</i>	268
EXPRESSING GENERALIZATION & SPECIALIZATION IN THE UML	268
A SIMPLE INHERITANCE EXAMPLE	269
<i>THE UML DIAGRAM</i>	269
<i>BASECLASS SOURCE CODE</i>	269
<i>DERIVEDCLASS SOURCE CODE</i>	270
<i>DRIVERAPPLICATION PROGRAM</i>	270
<i>Quick Review</i>	271
ANOTHER INHERITANCE EXAMPLE: PERSON - STUDENT	271
<i>THE PERSON - STUDENT UML CLASS DIAGRAM</i>	271
<i>PERSON - STUDENT SOURCE CODE</i>	272
<i>CASTING</i>	274
<i>USE CASTING SPARINGLY</i>	275
<i>Quick Review</i>	275
OVERRIDING BASE CLASS METHODS	275
<i>Quick Review</i>	276
ABSTRACT METHODS & ABSTRACT BASE CLASSES	276
<i>THE PRIMARY PURPOSE OF AN ABSTRACT BASE CLASS</i>	276
<i>EXPRESSING ABSTRACT BASE CLASSES IN UML</i>	277
<i>Quick Review</i>	279
INTERFACES	279
<i>THE PURPOSE OF INTERFACES</i>	279
<i>AUTHORIZED INTERFACE MEMBERS</i>	279
<i>THE DIFFERENCES BETWEEN AN INTERFACE AND AN ABSTRACT CLASS</i>	280
<i>EXPRESSING INTERFACES IN UML</i>	280
<i>EXPRESSING REALIZATION IN A UML CLASS DIAGRAM</i>	280
<i>AN INTERFACE EXAMPLE</i>	281
<i>Quick Review</i>	282
CONTROLLING HORIZONTAL & VERTICAL ACCESS	283
<i>Quick Review</i>	285
FINAL CLASSES & METHODS	285
<i>Quick Review</i>	285
POLYMORPHIC BEHAVIOR	285
<i>Quick Review</i>	285
INHERITANCE EXAMPLE: EMPLOYEE	286
INHERITANCE EXAMPLE: AIRCRAFT ENGINE SIMULATION	288
<i>AIRCRAFT ENGINE SIMULATION UML DIAGRAM</i>	288
<i>SIMULATION OPERATIONAL DESCRIPTION</i>	290
<i>TAKE A DEEP BREATH AND RELAX!</i>	290
<i>COMPILING THE AIRCRAFT ENGINE SIMULATION CODE</i>	290
COMPLETE AIRCRAFT ENGINE SIMULATION CODE LISTING	290
TERMS & DEFINITIONS	297
SUMMARY	297
SKILL-BUILDING EXERCISES	298
SUGGESTED PROJECTS	299
SELF-TEST QUESTIONS	300
REFERENCES	301
NOTES	302

PART III: GRAPHICAL USER INTERFACE PROGRAMMING

12 JAVA SWING API OVERVIEW

INTRODUCTION	306
AWT and SWING	306
<i>NAMING CONVENTIONS</i>	307
THE MATHEMATICS of GUIs	307
<i>COORDINATE SYSTEMS</i>	307
<i>COMPONENTS AND THEIR BOUNDS</i>	308
TOP-LEVEL CONTAINERS	308
<i>WINDOW AND JWINDOW</i>	309
<i>FRAME AND JFRAME</i>	309
<i>DIALOG AND JDialog</i>	309
<i>OUR FIRST GUI PROGRAM</i>	310
<i>EXPLANATION OF THE CODE</i>	311
<i>TOP-LEVEL CONTAINERS API</i>	312
<i>TOP-LEVEL CONTAINER CONSTRUCTORS</i>	312
<i>DIALOGUE WITH A SKEPTIC</i>	313
<i>TOP-LEVEL CONTAINER METHODS</i>	314
<i>QUICK REVIEW</i>	316
ORGANIZING COMPONENTS INTO CONTAINERS	316
<i>ABSOLUTE POSITIONING</i>	316
<i>LAYOUT MANAGERS</i>	318
<i>FlowLayout</i>	318
<i>GridLayout</i>	320
<i>BorderLayout</i>	321
<i>GridBagLayout</i>	323
<i>COMBINING LAYOUT MANAGERS USING JPanels</i>	327
<i>QUICK REVIEW</i>	328
THE COMPONENTS	329
<i>BRIEF DESCRIPTIONS</i>	329
<i>COMPONENT, CONTAINER, AND JComponent METHODS</i>	331
<i>Component Method Tables</i>	332
<i>Container Method Tables</i>	335
<i>JComponent Method Tables</i>	336
<i>QUICK REVIEW</i>	337
THE FINAL GUI	337
<i>HIGHLIGHTS OF THE FINAL GUI</i>	341
<i>LAYOUT OF THE FINAL GUI</i>	342
SUMMARY	344
SKILL-BUILDING EXERCISES	345
SUGGESTED PROJECTS	346
SELF-TEST QUESTIONS	346
REFERENCES	347
NOTES	347

13 HANDLING GUI EVENTS

INTRODUCTION	350
EVENT-HANDLING BASICS	350
<i>THE EVENT</i>	350
<i>THE EVENT LISTENER</i>	351
<i>REGISTERING AN EVENT LISTENER WITH A COMPONENT</i>	351

Using the API	352
Choosing the Right Event	352
Quick Review	354
CREATING EVENT LISTENERS	354
LISTENINGMainFrame0	354
LISTENINGMainFrame1	355
STEP 1: APPLICATION BEHAVIOR	355
STEP 2: THE SOURCE	355
STEP 3: THE EVENT	355
STEP 4: THE LISTENER	357
STEP 5: THE DESIGN APPROACH: EXTERNAL CLASS	358
STEP 6: REGISTRATION	358
LISTENINGMainFrame2	358
STEP 1: APPLICATION BEHAVIOR	359
STEP 2: THE SOURCE	359
STEP 3: THE EVENT	359
STEP 4: THE LISTENER	361
STEP 5: THE DESIGN APPROACH: EXTERNAL CLASS WITH FIELDS	362
STEP 6: REGISTRATION	363
LISTENINGMainFrame3	364
STEP 1: APPLICATION BEHAVIOR	364
STEP 2: THE SOURCE	364
STEP 3: THE EVENT	364
STEP 4: THE LISTENER	365
STEP 5: THE DESIGN APPROACH: INNER CLASS (FIELD-LEVEL)	366
STEP 6: REGISTRATION	367
LISTENINGMainFrame4	367
STEP 1: APPLICATION BEHAVIOR	367
STEP 2: THE SOURCE	367
STEP 3: THE EVENT	367
STEP 4: THE LISTENER	367
STEP 5: THE DESIGN APPROACH: INNER CLASS (LOCAL-VARIABLE-LEVEL)	368
STEP 6: REGISTRATION	368
LISTENINGMainFrame5	368
STEP 1: APPLICATION BEHAVIOR	369
STEP 2: THE SOURCE	369
STEP 3: THE EVENT	369
STEP 4: THE LISTENER	369
STEP 5: THE DESIGN APPROACH: EXISTING CLASS	369
STEP 6: REGISTRATION	371
LISTENINGMainFrame6	371
STEP 1: APPLICATION BEHAVIOR	371
STEP 2: THE SOURCE	371
STEP 3: THE EVENT	371
STEP 4: THE LISTENER	372
STEP 5: THE DESIGN APPROACH: ANONYMOUS CLASS	372
STEP 6: REGISTRATION	372
LISTENINGMainFrame7	373
STEP 1: APPLICATION BEHAVIOR	373
STEP 2: THE SOURCE	373
STEP 3: THE EVENT	373
STEP 4: THE LISTENER	373
STEP 5: THE DESIGN APPROACH: ANONYMOUS CLASS AND INSTANCE	374
STEP 6: REGISTRATION	374
ADAPTERS	374
SUMMARY	375
SKILL-BUILDING EXERCISES	376
SUGGESTED PROJECTS	376

SELF-TEST QUESTIONS	377
REFERENCES	377
NOTES	377

14 AN ADVANCED GUI PROJECT

INTRODUCTION	380
<i>THE PROJECT</i>	380
<i>THE APPROACH</i>	380
STEP 1 – LAYING THE GROUNDWORK	381
<i>THE GARMENT CLASS</i>	381
<i>THE DRESSINGBOARD CLASS</i>	382
<i>GRAPHICS AND THE AWT/SWING FRAMEWORK</i>	382
<i>GRAPHICS (THE “HOW” OF DRAWING)</i>	382
<i>THE AWT/SWING FRAMEWORK (THE “WHEN” OF DRAWING)</i>	384
<i>CALLING REPAINT</i>	385
<i>OBTAINING A GRAPHICS2D</i>	386
<i>THE DRESSINGBOARD CLASS (CONTINUED)</i>	386
<i>LOADING AN IMAGE OR RESOURCE</i>	387
<i>THE MAINFRAME CLASS</i>	388
<i>QUICK REVIEW</i>	389
STEP 2 – SWING’S SEPARABLE MODEL ARCHITECTURE	389
<i>USING A LISTMODEL</i>	389
<i>THE MAINFRAME CLASS</i>	390
<i>QUICK REVIEW</i>	392
STEP 3 – WRITING A CUSTOM RENDERER	392
<i>USING A RENDERER</i>	393
<i>THE CHECKBOXLISTCELL CLASS</i>	394
<i>THE MAINFRAME CLASS</i>	396
<i>QUICK REVIEW</i>	398
STEP 4 – THE ART OF ILLUSION	398
<i>THE CHECKBOXLISTCELL CLASS</i>	401
<i>THE MAINFRAME CLASS</i>	402
<i>QUICK REVIEW</i>	403
STEP 5 – DEFINING YOUR OWN EVENTS AND EVENTLISTENERS	403
<i>THE CHECKBOXLISTCELL CLASS</i>	403
<i>THE MAINFRAME CLASS</i>	406
<i>QUICK REVIEW</i>	408
INTERLUDE – WRITING A CUSTOM EDITOR	408
<i>THE DEMOTREEORTABLECELLHANDLER CLASS</i>	410
<i>THE DEMOFRAME CLASS</i>	412
<i>QUICK REVIEW</i>	413
STEP 6 – A GOOD COMPONENT GETS BETTER	413
<i>THE DRAGLIST CLASS – OVERVIEW</i>	414
<i>ALGORITHM</i>	414
<i>USE DEFAULTLISTMODEL (LINES 41 - 65)</i>	414
<i>REORDEREVENT AND REORDERLISTENER (LINES 184 - 203)</i>	414
<i>THE DRAGLIST CLASS – DRAGGING</i>	415
<i>DRAGINDEX, DRAGITEM AND DRAGRECT ATTRIBUTES</i>	415
<i>DRAGSTART, DRAGTHRESHOLD AND ALLOWDRAG ATTRIBUTES</i>	415
<i>DELTA Y AND INDRAG ATTRIBUTES</i>	415
<i>HOW THE INDRAG ATTRIBUTE IS USED</i>	416
<i>THE DRAGLIST CLASS – PAINTING</i>	416
<i>THE CREATEDRAGIMAGE() METHOD (LINES 66 - 94)</i>	416
<i>THE PAINTCOMPONENT() METHOD (LINES 95 - 108)</i>	417
<i>THE DRAGLIST CLASS – CODE</i>	417
<i>THE MAINFRAME CLASS</i>	419

SUMMARY	421
SKILL-BUILDING EXERCISES	422
SUGGESTED PROJECTS	422
SELF-TEST QUESTIONS	423
REFERENCES	423
NOTES	423

PART IV: INTERMEDIATE CONCEPTS

15 EXCEPTIONS

INTRODUCTION	428
THE THROWABLE CLASS HIERARCHY	428
<i>Errors</i>	429
<i>Exceptions</i>	429
<i>RuntimeExceptions</i>	429
<i>Checked vs. Unchecked Exceptions</i>	430
<i>Quick Review</i>	430
HANDLING EXCEPTIONS	430
<i>Try/Catch Statement</i>	431
<i>Multiple Catch Blocks</i>	431
<i>The Importance Of Exception Type And Order In Multiple Catch Blocks</i>	432
<i>Manipulating A Throwable Object</i>	433
<i>Try/Catch/Finally Statement</i>	435
<i>Try/Finally Statement</i>	435
<i>Quick Review</i>	436
THROWING EXCEPTIONS	436
<i>The Throws Clause</i>	436
<i>The Throw Keyword</i>	437
<i>Quick Review</i>	438
CREATING CUSTOM EXCEPTIONS	438
<i>High & Low Level Abstractions</i>	438
<i>Extending The Exception Class</i>	438
<i>Some Advice On Naming Your Custom Exceptions</i>	440
<i>Quick Review</i>	440
SUMMARY	440
SKILL-BUILDING EXERCISES	441
SUGGESTED PROJECTS	441
SELF-TEST QUESTIONS	441
REFERENCES	441
NOTES	442

16 THREADS

INTRODUCTION	444
WHAT IS A THREAD?	444
<i>Quick Review</i>	446
BUILDING A CLOCK COMPONENT	446
<i>currentThread(), sleep(), interrupt(), interrupted() and isInterrupted()</i>	447
<i>Creating Your Own Threads</i>	449
<i>Quick Review</i>	451

Computing Pi	452
<i>Quick Review</i>	455
Thread Priority and Scheduling	455
<i>Your Results May Vary</i>	457
<i>Quick Review</i>	458
Race Conditions	459
<i>The Mechanics of Synchronization</i>	460
<i>Three Basic Rules of Synchronization</i>	462
<i>Synchronization Rule 1.....</i>	462
<i>Synchronization Rule 2.....</i>	463
<i>Synchronization Rule 3.....</i>	465
<i>Synchronizing Methods</i>	466
<i>Quick Review</i>	467
The Producer-Consumer Relationship	468
<i>Quick Review</i>	473
Deadlock	474
<i>Quick Review</i>	476
About the Pi-Generating Algorithm	477
Summary	477
Skill-Building Exercises	478
Suggested Projects	479
Self-Test Questions	479
References	480
Notes	480

17 Collections

Introduction	482
Case Study: Building A Dynamic Array	482
<i>Evaluating DynamicArray</i>	484
<i>The ArrayList Class To The Rescue</i>	484
<i>The Power Of Collections – Polymorphic Code</i>	485
Java Collections Framework Overview	486
<i>The Purpose Of The Collections Framework</i>	486
<i>Framework Organization</i>	486
<i>Core Collections Interfaces.....</i>	486
<i>General Purpose Implementation Classes.....</i>	487
<i>Mapping An Implementation Class To Its Underlying Data Structure.....</i>	492
<i>Algorithms.....</i>	492
<i>Infrastructure.....</i>	493
<i>Quick Review</i>	493
Java 1.4.2 Style Collections	493
<i>Creating A Set From A List</i>	494
<i>PeopleManager Revisited</i>	494
<i>Casting Retrieved Objects</i>	496
<i>Creating New Data Structures From Existing Collections</i>	497
<i>Quick Review</i>	499
Java 5 Style Collections: Generics	499
<i>Java 5 Collection Framework Core Interfaces</i>	499
<i>Java 5 Collections Framework Sample Programs</i>	500
<i>SetTestApp Program Revised.....</i>	500
<i>When To Use The Enhanced For Loop.....</i>	501
<i>Static Polymorphism – Generic Methods</i>	501
<i>Quick Review</i>	502
<i>Moving Forward</i>	502
Summary	503

Skill-Building Exercises	503
Suggested Projects	504
Self-Test Questions	504
References	505
Notes	505

18 File I/O

Introduction	508
JAVA I/O PACKAGE OVERVIEW	508
<i>File</i> Class	508
<i>InputStream</i> Classes	508
<i>OutputStream</i> Classes	510
Reader Classes	510
Writer Classes	510
<i>RandomAccessFile</i> Class	510
How Do You Choose Between Byte Streams, Character Streams, and <i>RandomAccessFile</i> ?	510
Another Way To Categorize The Java I/O Classes	510
Quick Review	512
Using The File Class	513
Quick Review	515
SEQUENTIAL BYTE STREAM FILE I/O USING INPUT- & OUTPUTSTREAMS	515
<i>OutputStreams</i>	515
<i>FileOutputStream</i>	515
<i>BufferedOutputStream</i>	516
<i>DataOutputStream</i>	517
<i>ObjectOutputStream</i>	518
<i>PrintStream</i>	520
<i>InputStreams</i>	520
<i>FileInputStream</i>	521
<i>BufferedInputStream</i>	521
<i>DataInputStream</i>	522
<i>ObjectInputStream</i>	523
Quick Review	523
SEQUENTIAL CHARACTER STREAM FILE I/O USING READERS & WRITERS	524
Writers	524
<i>FileWriter</i>	524
<i>BufferedWriter</i>	525
<i>OutputStreamWriter</i>	526
<i>PrintWriter</i>	526
Readers	527
<i>FileReader</i>	527
<i>InputStreamReader</i>	527
<i>BufferedReader</i>	528
Quick Review	529
A PRACTICAL FILE I/O EXAMPLE: THE java.util.PROPERTIES CLASS	529
Quick Review	531
RANDOM ACCESS FILE I/O	531
Towards An Approach To The Adapter Project	533
Start Small And Take Baby Steps.....	533
Other Project Considerations	534
Locking A Record For Updates And Deletes	534
Translating Low-Level Exceptions Into Higher-Level Exception Abstractions.....	534
Where To Go From Here	534
Complete <i>RandomAccessFile</i> Legacy <i>Datafile</i> Adapter Source Code Listing	535
Quick Review	544

Quick File I/O COMBINATION REFERENCE	545
SUMMARY	546
SKILL-BUILDING EXERCISES	547
SUGGESTED PROJECTS	547
SELF-TEST QUESTIONS	548
REFERENCES	549
NOTES	549

PART V: NETWORK PROGRAMMING

19 INTRODUCTION TO NETWORKING AND DISTRIBUTED APPLICATIONS

INTRODUCTION	554
WHAT IS A COMPUTER NETWORK?	554
<i>PURPOSE OF A NETWORK</i>	554
<i>THE ROLE OF NETWORK PROTOCOLS</i>	555
<i>HOMOGENEOUS VS. HETEROGENEOUS NETWORKS</i>	555
<i>THE UNIFYING NETWORK PROTOCOLS: TCP/IP</i>	555
<i>WHAT'S SO SPECIAL ABOUT THE INTERNET?</i>	556
<i>QUICK REVIEW</i>	556
SERVERS & CLIENTS	557
<i>SERVER HARDWARE AND SOFTWARE</i>	557
<i>CLIENT HARDWARE AND SOFTWARE</i>	557
<i>QUICK REVIEW</i>	558
APPLICATION DISTRIBUTION	558
<i>PHYSICAL DISTRIBUTION ON ONE COMPUTER</i>	558
<i>RUNNING MULTIPLE JAVA VIRTUAL MACHINES ON THE SAME COMPUTER</i>	558
<i>STARTING MULTIPLE JVMs IN WINDOWS</i>	558
<i>STARTING MULTIPLE JVMs IN UNIX AND LINUX</i>	559
<i>RUNNING MULTIPLE CLIENTS ON THE SAME COMPUTER</i>	561
<i>ADDRESSING THE LOCAL MACHINE</i>	561
<i>PHYSICAL DISTRIBUTION ACROSS MULTIPLE COMPUTERS</i>	561
<i>MULTIPLE JAVA VIRTUAL MACHINES</i>	562
<i>QUICK REVIEW</i>	562
MULTI-TIERED APPLICATIONS	562
<i>LOGICAL APPLICATION TIERS</i>	562
<i>PHYSICAL TIER DISTRIBUTION</i>	563
<i>QUICK REVIEW</i>	564
INTERNET NETWORKING PROTOCOLS – NUTS & BOLTS	564
<i>THE INTERNET PROTOCOLS: TCP, UDP, AND IP</i>	564
<i>THE APPLICATION LAYER</i>	564
<i>TRANSPORT LAYER</i>	565
<i>NETWORK LAYER</i>	565
<i>DATA LINK & PHYSICAL LAYERS</i>	565
<i>PUTTING IT ALL TOGETHER</i>	566
<i>WHAT YOU NEED TO KNOW</i>	566
<i>QUICK REVIEW</i>	566
JAVA SUPPORT FOR NETWORK PROGRAMMING	567
<i>A NETWORK PROGRAMMING SCENARIO USING SOCKETS – OVERVIEW</i>	567
<i>THE URL CLASS</i>	568
<i>QUICK REVIEW</i>	569
REMOTE METHOD INVOCATION (RMI) OVERVIEW	569
<i>CREATING & RUNNING AN RMI APPLICATION IN JAVA 1.4.2</i>	570

REQUIRED STEPS	570
STEP 1: DEFINE A REMOTE INTERFACE.....	570
STEP 2: DEFINE A REMOTE INTERFACE IMPLEMENTATION.....	571
STEP 3: COMPILE THE REMOTE INTERFACE & REMOTE INTERFACE IMPLEMENTATION SOURCE FILES.....	571
STEP 4: USE THE RMIC TOOL TO GENERATE STUB CLASS	571
STEP 5: CREATE SERVER APPLICATION.....	572
STEP 6: CREATE CLIENT APPLICATION.....	573
STEP 7: START THE RMI REGISTRY.....	573
STEP 8: RUN THE SERVER APPLICATION.....	574
STEP 9: RUN THE CLIENT APPLICATION.....	574
CREATING & RUNNING AN RMI APPLICATION IN JAVA 1.5	575
CONSIDERATIONS WHEN DEPLOYING RMI APPLICATIONS IN MIXED JVM VERSION ENVIRONMENTS	575
SERVER JVM 1.5 - CLIENT JVM 1.5.....	575
SERVER JVM 1.5 - CLIENT JVM 1.4.2.....	575
SERVER JVM 1.4.2 - CLIENT JVM 1.5.....	575
SERVER JVM 1.4.2 - CLIENT JVM 1.4.2.....	575
Quick Review	576
SUMMARY	576
SKILL-BUILDING EXERCISES	577
SUGGESTED PROJECTS	578
SELF-TEST QUESTIONS	578
REFERENCES	579
NOTES	579

20 CLIENT-SERVER APPLICATIONS

INTRODUCTION	582
A SIMPLE SOCKET-BASED CLIENT-SERVER EXAMPLE	582
Simple SERVER Application	583
Simple CLIENT Application	585
RUNNING THE EXAMPLES	586
A FEW WORDS ABOUT PROTOCOL	588
Quick Review	588
PROJECT SPECIFICATION	588
CLIENT-SERVER PROJECT INITIAL DESIGN CONSIDERATIONS	589
NOUN-VERB ANALYSIS	589
HIGH-LEVEL APPLICATION OPERATIONS	590
HOW TO PROCEED	592
Quick Review	592
IMPLEMENTING NETRATSERVER – FIRST ITERATION	592
FIRST ITERATION CODE	593
TESTING THE FIRST ITERATION CODE	595
Quick Review	596
IMPLEMENTING NETRATSERVER - SECOND ITERATION	596
SECOND ITERATION CODE	597
TESTING SECOND ITERATION CODE	599
Quick Review	599
THE RMI-BASED CLIENT	600
TESTING MULTIPLE RMI CLIENTS	602
Quick Review	602
NETRATSERVER IMPLEMENTATION – THIRD ITERATION	602
UPDATED SERVER APPLICATION CLASS DIAGRAM	602
THIRD ITERATION CODE	603
TESTING THE THIRD ITERATION CODE	604
TESTING MULTIPLE RMI CLIENT APPLICATIONS	606
Quick Review	607

NETRATSERVER IMPLEMENTATION – FOURTH ITERATION	607
<i>FOURTH ITERATION CODE</i>	<i>608</i>
<i>TESTING FOURTH ITERATION CODE</i>	<i>610</i>
<i>TESTING THE FOURTH ITERATION CODE</i>	<i>613</i>
<i>QUICK REVIEW</i>	<i>613</i>
NETRATSERVER IMPLEMENTATION – FINAL ITERATION	613
<i>FINAL ITERATION CODE</i>	<i>615</i>
<i>TESTING THE FINAL ITERATION CODE</i>	<i>619</i>
<i>PARTING COMMENTS</i>	<i>619</i>
<i>QUICK REVIEW</i>	<i>620</i>
SUMMARY	620
SKILL-BUILDING EXERCISES	621
SUGGESTED PROJECTS	622
SELF-TEST QUESTIONS	622
REFERENCES	623
NOTES	623

21 Applets & JDBC

INTRODUCTION	626
APPLET OVERVIEW	626
<i>THE BENEFITS OF USING APPLETS</i>	<i>626</i>
<i><APPLET> VS. <OBJECT> TAGS</i>	<i>627</i>
<i>BROWSER JVM VS. JAVA PLUG-IN</i>	<i>627</i>
A BASIC APPLET EXAMPLE	627
<i>APPLET CODE</i>	<i>627</i>
<i>HTML PAGE CODE</i>	<i>628</i>
<i>PACKAGING AND DISTRIBUTION</i>	<i>628</i>
<i>RUNNING THE BASIC APPLET EXAMPLE</i>	<i>628</i>
<i>APPLET LIFE-CYCLE STAGES.....</i>	<i>629</i>
<i>THE <APPLET> TAG IN DETAIL</i>	<i>630</i>
<i>QUICK REVIEW</i>	<i>631</i>
APPLET SECURITY RESTRICTIONS	631
<i>MISCELLANEOUS APPLET SECURITY RESTRICTIONS</i>	<i>634</i>
<i>SECURITY POLICIES & SIGNED APPLETS</i>	<i>634</i>
<i>QUICK REVIEW</i>	<i>634</i>
USING APPLET PARAMETERS	634
<i>QUICK REVIEW</i>	<i>636</i>
EXTENDED APPLET EXAMPLE – POETRY IN BROWSER	636
JAVA DATABASE CONNECTIVITY (JDBC™) OVERVIEW	639
<i>JDBC™ PACKAGES</i>	<i>639</i>
<i>JDBC™ SPECIFICATION</i>	<i>639</i>
<i>USING JDBC – A MACRO VIEW</i>	<i>640</i>
<i>PROPERLY CONFIGURED DATABASE.....</i>	<i>640</i>
<i>JDBC DRIVER CLASS.....</i>	<i>640</i>
<i>KNOWLEDGE OF RELATIONAL DATABASE DESIGN</i>	<i>640</i>
<i>KNOWLEDGE OF THE DATABASE SCHEMA</i>	<i>640</i>
<i>KNOWLEDGE OF SQL.....</i>	<i>640</i>
<i>THE JAVA JDBC PROGRAM</i>	<i>640</i>
<i>GENERAL STEPS REQUIRED TO EMPLOY JDBC</i>	<i>641</i>
<i>MOVING FORWARD</i>	<i>641</i>
<i>QUICK REVIEW</i>	<i>641</i>
JDBC PROJECT DESCRIPTION	641
<i>OVERALL SYSTEM ARCHITECTURE</i>	<i>641</i>
<i>DETAILED SYSTEM ARCHITECTURE</i>	<i>641</i>
<i>ROLE OF THE PERSISTENT CLASS</i>	<i>643</i>

<i>Quick Review</i>	643
MySQL DATABASE & JDBC	643
<i>Setting Up MySQL To Run The Example Code</i>	643
<i>Obtaining and Installing MySQL</i>	644
<i>The mysqladmin Program</i>	644
<i>The mysql Monitor Program</i>	645
<i>Helpful MySQL Monitor Commands</i>	646
<i>Creating The CHAPTER_21 Database</i>	647
<i>Establishing Database Security Via MySQL Access Control Tables</i>	647
<i>General Strategy For Managing Permissions Using The MySQL Access Control Tables</i>	649
<i>Creating The Employee Training Management System Tables</i>	650
<i>Populating Tables with Data and Running Queries</i>	651
<i>Inserting Data Into Tables</i>	651
<i>Querying A Table with the select Statement</i>	652
<i>Updating Data In A Table</i>	652
<i>Joining Tables With A Select Statement</i>	652
<i>Accessing CHAPTER_21 Database Tables Via JDBC</i>	654
<i>A Short JDBC Example Program</i>	654
<i>Accessing ResultSet Metadata</i>	655
<i>Quick Review</i>	656
Extended Applet & JDBC Example	656
<i>The Code</i>	656
<i>Compiling And Packaging The Applet Code For Distribution</i>	667
<i>Running The Example</i>	668
<i>Start MySQL Database Application</i>	668
<i>Start The DBServerApp Program</i>	668
<i>Access The employeetraining.html Page</i>	668
<i>Using The Employee Training Applet</i>	668
<i>Code Discussion</i>	670
<i>Employee and EmployeeTraining Classes</i>	670
<i>AddNewEmployeeDialog and AddTrainingRecordDialog Classes</i>	671
<i>EmployeeTrainingApplet Class</i>	671
<i>DBServerApp Class</i>	671
<i>Persister Class</i>	671
<i>DBServerProperties</i>	671
<i>Statements vs. Prepared Statements</i>	672
SUMMARY	672
Skill-Building Exercises	673
SUGGESTED PROJECTS	674
Self-Test Questions	675
REFERENCES	675
NOTES	676

PART VI: OBJECT-ORIENTED DESIGN

22 INHERITANCE, COMPOSITION, INTERFACES, POLYMORPHISM

INTRODUCTION	680
INHERITANCE VS. COMPOSITION: THE GREAT DEBATE	680
<i>What's The End Game?</i>	681
<i>Flexible Application Architectures</i>	681
<i>Modularity And Reliability</i>	681
<i>Architectural Stability Via Managed Dependencies</i>	681

<i>Knowing When To Accept A Design That's Good Enough</i>	682
<i>Quick Review</i>	682
INHERITANCE-BASED DESIGN	682
<i>THREE GOOD REASONS TO USE INHERITANCE</i>	682
<i>AS A MEANS TO REASON ABOUT CODE BEHAVIOR</i>	682
<i>TO GAIN A MEASURE OF CODE REUSE</i>	682
<i>TO FACILITATE INCREMENTAL DEVELOPMENT</i>	682
<i>FORMS OF INHERITANCE: MEYER'S INHERITANCE TAXONOMY</i>	683
<i>COAD'S INHERITANCE CRITERIA</i>	684
<i>PERSON - EMPLOYEE EXAMPLE REVISITED</i>	685
<i>Quick Review</i>	685
THE ROLE OF INTERFACES	686
<i>REDUCING OR LIMITING INTERMODULE DEPENDENCIES</i>	686
<i>MODELING DOMINANT, COLLATERAL, AND DYNAMIC ROLES</i>	686
<i>DOMINANT ROLES</i>	687
<i>COLLATERAL ROLES</i>	687
<i>DYNAMIC ROLES</i>	687
<i>Quick Review</i>	687
Applied Polymorphism	687
<i>Quick Review</i>	688
Composition-Based Design As A Force Multiplier	688
<i>TWO TYPES OF AGGREGATION</i>	688
<i>POLYMORPHIC CONTAINMENT</i>	688
<i>AN EXTENDED EXAMPLE</i>	689
<i>Quick Review</i>	693
SUMMARY	693
Skill-Building Exercises	694
SUGGESTED PROJECTS	695
SELF-TEST QUESTIONS	696
REFERENCES	696
NOTES	697

23 Well-Behaved Objects

INTRODUCTION	700
CONSIDER OBJECT USAGE FROM THE START	700
<i>Applying THE OBJECT USAGE SCENARIO CHECKLIST</i>	701
<i>Quick Review</i>	702
OVERRIDING java.lang.Object METHODS	703
<i>OVERRIDING toString()</i>	703
<i>OVERRIDING equals() AND hashCode()</i>	703
<i>equals() METHOD</i>	703
<i>TESTING PERSON.toString() AND PERSON.equals() METHODS</i>	705
<i>hashCode() METHOD</i>	705
<i>OVERRIDING hashCode() IN THE PERSON CLASS</i>	707
<i>TESTING PERSON.hashCode()</i>	707
<i>OVERRIDING clone()</i>	708
<i>SHALLOW COPY VS. DEEP COPY</i>	708
<i>OVERRIDING PERSON.clone()</i>	709
<i>OVERRIDING finalize() – AND AN ALTERNATIVE APPROACH</i>	710
<i>Quick Review</i>	710
IMPLEMENTING THE java.lang.Comparable INTERFACE	711
<i>Quick Review</i>	712
Using PERSON OBJECTS IN A COLLECTION	713
IMPLEMENTING java.util.Comparator INTERFACE	713
<i>Quick Review</i>	715

IMPLEMENTING THE SERIALIZABLE INTERFACE	716
PERSON CLASS – THE FINAL VERSION	716
SUMMARY	717
SKILL-BUILDING EXERCISES	718
SUGGESTED PROJECTS	718
SELF-TEST QUESTIONS	718
REFERENCES	719
NOTES	719

24 THREE DESIGN PRINCIPLES

INTRODUCTION	722
THE PREFERRED CHARACTERISTICS OF AN OBJECT-ORIENTED ARCHITECTURE	722
<i>EASY TO UNDERSTAND – (HOW DOES THIS THING WORK?)</i>	722
<i>EASY TO REASON ABOUT – (WHAT ARE THE EFFECTS OF CHANGE?)</i>	722
<i>EASY TO EXTEND – (WHERE DO I ADD FUNCTIONALITY?)</i>	722
THE LISKOV SUBSTITUTION PRINCIPLE & DESIGN BY CONTRACT	723
<i>REASONING ABOUT THE BEHAVIOR OF SUPERTYPES AND SUBTYPES</i>	723
<i>RELATIONSHIP BETWEEN THE LSP AND DbC</i>	723
<i>THE COMMON GOAL OF THE LSP AND DbC</i>	723
<i>JAVA SUPPORT FOR THE LSP AND DbC</i>	723
<i>DESIGNING WITH THE LSP/DbC IN MIND</i>	724
<i>CLASS DECLARATIONS VIEWED AS BEHAVIOR SPECIFICATIONS</i>	724
<i>Quick Review</i>	724
PRECONDITIONS, POSTCONDITIONS, AND CLASS INVARIANTS	724
<i>CLASS INVARIANT</i>	724
<i>PRECONDITION</i>	724
<i>POSTCONDITION</i>	725
<i>AN EXAMPLE</i>	725
<i>A NOTE ON USING ASSERTIONS TO ENFORCE PRE- AND POSTCONDITIONS</i>	726
<i>USING INCREMENTER AS A BASE CLASS</i>	726
<i>CHANGING THE PRECONDITIONS OF DERIVED CLASS METHODS</i>	728
<i>CHANGING THE POSTCONDITIONS OF DERIVED CLASS METHODS</i>	731
<i>SPECIAL CASES OF PRECONDITIONS AND POSTCONDITIONS</i>	732
<i>METHOD ARGUMENT TYPES</i>	732
<i>METHOD RETURN TYPES</i>	735
<i>THREE RULES OF THE SUBSTITUTION PRINCIPLE</i>	735
<i>SIGNATURE RULE</i>	735
<i>METHODS RULE</i>	735
<i>PROPERTIES RULE</i>	735
<i>Quick Review</i>	735
THE OPEN-CLOSED PRINCIPLE	735
<i>ACHIEVING THE OPEN-CLOSED PRINCIPLE</i>	736
<i>AN OCP EXAMPLE</i>	736
<i>Quick Review</i>	739
THE DEPENDENCY INVERSION PRINCIPLE	740
<i>CHARACTERISTICS OF BAD SOFTWARE ARCHITECTURE</i>	740
<i>CHARACTERISTICS OF GOOD SOFTWARE ARCHITECTURE</i>	741
<i>SELECTING THE RIGHT ABSTRACTIONS TAKES EXPERIENCE</i>	741
<i>Quick Review</i>	741
TERMS AND DEFINITIONS	741
SUMMARY	742
SKILL-BUILDING EXERCISES	743
SUGGESTED PROJECTS	743
SELF-TEST QUESTIONS	743
REFERENCES	744

NOTES	744
-------------	-----

25 Helpful Design PATTERNS

INTRODUCTION	746
SOFTWARE DESIGN PATTERNS AND HOW THEY CAME TO BE	746
<i>WHAT EXACTLY IS A SOFTWARE DESIGN PATTERN?</i>	746
<i>ORIGINS</i>	746
<i>PATTERN SPECIFICATION</i>	747
<i>APPLYING SOFTWARE DESIGN PATTERNS</i>	747
<i>QUICK REVIEW</i>	747
THE SINGLETON PATTERN	748
<i>QUICK REVIEW</i>	749
THE FACTORY PATTERN	749
<i>THE DYNAMIC FACTORY</i>	750
<i>ADVANTAGES OF THE DYNAMIC FACTORY PATTERN</i>	751
<i>QUICK REVIEW</i>	751
THE MODEL-VIEW-CONTROLLER PATTERN	751
<i>POTENTIAL ISSUES WITH ONE ACTIONLISTENER</i>	753
<i>QUICK REVIEW</i>	753
THE COMMAND PATTERN	753
<i>QUICK REVIEW</i>	756
A COMPREHENSIVE PATTERN-BASED EXAMPLE	756
<i>Code Listing By Package Name</i>	756
<i>com.pulpfreepress.commands</i>	756
<i>com.pulpfreepress.controller</i>	759
<i>com.pulpfreepress.exceptions</i>	760
<i>com.pulpfreepress.interfaces</i>	760
<i>com.pulpfreepress.model</i>	761
<i>com.pulpfreepress.utils</i>	767
<i>com.pulpfreepress.view</i>	769
SUMMARY	776
SKILL-BUILDING EXERCISES	777
SUGGESTED PROJECTS	777
SELF-TEST QUESTIONS	777
REFERENCES	778
NOTES	778

APPENDICES

Appendix A: Helpful Checklists And Tables

PROJECT-APPROACH STRATEGY CHECK-OFF LIST	781
DEVELOPMENT CYCLE	782

Appendix B: ASCII Table

ASCII Table	783
--------------------------	------------

List of Tables

Table 1-1: JAVA SOURCE FILE RULES SUMMARY	17
Table 1-2: CLASS NAMING EXAMPLES	21
Table 1-3: CONSTANT NAMING EXAMPLES	21
Table 1-4: VARIABLE NAMING EXAMPLES	21
Table 1-5: METHOD NAMING EXAMPLES	22
Table 2-1: HELPFUL OPERATING SYSTEM COMMANDS	56
Table 3-1: PROJECT APPROACH STRATEGY	60
Table 3-2: DEVELOPMENT CYCLE	61
Table 3-3: PROJECT SPECIFICATION	62
Table 3-4: ROBOT RAT NOUNS AND VERBS	64
Table 3-5: LANGUAGE FEATURE STUDY CHECK-OFF LIST FOR ROBOT RAT PROJECT	66
Table 3-6: FIRST ITERATION DESIGN CONSIDERATIONS	69
Table 3-7: SECOND ITERATION DESIGN CONSIDERATIONS	71
Table 3-8: THIRD ITERATION DESIGN CONSIDERATIONS	72
Table 3-9: FOURTH ITERATION DESIGN CONSIDERATIONS	76
Table 3-10: FIFTH ITERATION DESIGN CONSIDERATIONS	80
Table 3-11: FINAL PROJECT REVIEW CHECKLIST	83
Table 5-1: JAVA PLATFORM PACKAGES USED IN THIS BOOK	116
Table 6-1: TERMS AND DEFINITIONS TO GET YOU STARTED	122
Table 6-2: RESERVED JAVA KEYWORDS	127
Table 6-3: JAVA PRIMITIVE DATA TYPES	129
Table 6-4: JAVA OPERATORS	139
Table 6-5: PRIMITIVE TYPE WRAPPER CLASSES	149
Table 7-1: JAVA SELECTION AND ITERATION STATEMENT SELECTION GUIDE	171
Table 8-1: JAVA ARRAY PROPERTIES	183
Table 8-2: JAVA API CLASSES AND INTERFACES REFERENCED IN CHAPTER 8	202
Table 8-3: EISCS MACHINE INSTRUCTIONS	205
Table 9-1: PEOPLE MANAGER PROGRAM CLASS RESPONSIBILITIES	212
Table 9-2: JAVA METHOD MODIFIER KEYWORDS	218
Table 11-1: DIFFERENCES BETWEEN ABSTRACT CLASSES AND INTERFACES	280
Table 11-2: CHAPTER 11 TERMS AND DEFINITIONS	297
Table 12-1: JAVA.AWT.COMPONENT.SETBOUNDS() METHODS	311
Table 12-2: TOP-LEVEL CONTAINERS CONSTRUCTOR CHART	312
Table 12-3: METHODS AVAILABLE TO ALL DESCENDANTS OF WINDOW	314
Table 12-4: METHODS AVAILABLE TO FRAME, JFRAME, Dialog, JDialog ONLY	314
Table 12-5: METHODS AVAILABLE TO FRAME, JFRAME ONLY	315
Table 12-6: METHODS AVAILABLE TO JWindow, JFRAME, JDialog ONLY	315
Table 12-7: METHODS AVAILABLE TO JFRAME, JDialog ONLY	315
Table 12-8: GridBagConstraints Fields and Their Default Values	326
Table 12-9: TOP-LEVEL COMPONENTS FOR CONTAINING OTHER COMPONENTS	330
Table 12-10: NON TOP-LEVEL COMPONENTS FOR CONTAINING OTHER COMPONENTS	330
Table 12-11: COMPONENTS THAT ALLOW THE SELECTION OF A VALUE FROM A DISCRETE SET OF VALUES	330
Table 12-12: COMPONENTS THAT ALLOW THE SELECTION OF A VALUE FROM A VIRTUAL CONTINUUM OF VALUES	330
Table 12-13: COMPONENTS THAT ALLOW THE USER TO INITIATE AN ACTION	331
Table 12-14: COMPONENTS THAT REPRESENT A BOOLEAN VALUE	331
Table 12-15: COMPONENTS FOR ENTERING TEXT	331
Table 12-16: VIEW-ONLY COMPONENTS	331
Table 12-17: APPEARANCE-RELATED COMPONENT METHODS	332
Table 12-18: SIZE- AND LOCATION-RELATED COMPONENT METHODS	333
Table 12-19: VISIBILITY-RELATED COMPONENT METHODS	334

Table 12-20: CONTAINMENT HIERARCHY-RELATED COMPONENT METHODS	334
Table 12-21: OTHER PROPERTY-RELATED COMPONENT METHODS	334
Table 12-22: APPEARANCE-RELATED CONTAINER METHODS	335
Table 12-23: CONTAINMENT HIERARCHY-RELATED CONTAINER METHODS	335
Table 12-24: APPEARANCE-RELATED JCOMPONENT METHODS	336
Table 12-25: SIZE- AND LOCATION-RELATED JCOMPONENT METHODS	336
Table 12-26: VISIBILITY-RELATED JCOMPONENT METHODS	336
Table 12-27: CONTAINMENT HIERARCHY-RELATED JCOMPONENT METHODS	337
Table 12-28: OTHER PROPERTY-RELATED JCOMPONENT METHODS	337
Table 13-1: EVENTOBJECT METHODS	351
Table 13-2: COMPONENT METHODS FOR MANAGING EVENT LISTENERS	351
Table 13-3: JBUTTON'S EVENTLISTENER REGISTRATION METHODS	352
Table 13-4: LISTENERS AND EVENT TYPES FOR JMENUITEM	355
Table 13-5: ACTIONEVENT EVENT IDS	356
Table 13-6: ACTIONEVENT CONSTANTS	356
Table 13-8: ACTIONLISTENER METHODS	357
Table 13-7: ACTIONEVENT PROPERTIES	357
Table 13-9: COMPONENTEVENT METHODS	359
Table 13-10: INPUTEVENT METHODS	359
Table 13-11: INPUTEVENT CONVENIENCE METHODS	359
Table 13-12: MOUSEEVENT EVENT IDS	360
Table 13-13: MOUSEEVENT-SPECIFIC METHODS	360
Table 13-14: SWINGUTILITIES HELPER METHODS	361
Table 13-15: MOUSELISTENER METHODS	361
Table 13-16: MOUSEMOTIONLISTENER METHODS	362
Table 13-17: MOUSEWHEELLISTENER METHODS	362
Table 13-18: KEYEVENT EVENT IDS	365
Table 13-19: KEYEVENT METHODS	365
Table 13-20: KEYLISTENER METHODS	366
Table 13-21: CHANGETLISTENER METHODS	372
Table 13-22: LISTSELECTIONEVENT METHODS	373
Table 13-23: LISTSELECTIONLISTENER METHODS	374
Table 14-1: COMPONENT'S PAINTING METHOD	384
Table 14-2: JCOMPONENT'S PAINTING METHODS	385
Table 14-3: REPAINT METHODS DEFINED BY COMPONENT	385
Table 14-4: REPAINT METHODS DEFINED BY JCOMPONENT	386
Table 14-5: LISTMODEL METHODS	390
Table 14-6: JLIST'S LISTMODEL METHODS	390
Table 14-7: JLIST'S, JTREE'S AND JCOMBOBOX'S RENDERER-RELATED METHODS	393
Table 14-8: JTABLE'S RENDERER-RELATED METHODS	393
Table 14-9: JTREE'S EDITOR-RELATED METHODS	409
Table 14-10: JTABLE'S AND TABLECOLUMN'S EDITOR-RELATED METHODS	409
Table 14-11: JAVAX.SWING.CELLEDITOR METHODS	409
Table 14-12: DRAGINDEX, DRAGITEM AND DRAGRECT ATTRIBUTES	415
Table 14-13: DRAGSTART, DRAGTHRESHOLD AND ALLOWDRAG ATTRIBUTES	415
Table 14-14: DELTAY AND INDRAG ATTRIBUTES	416
Table 14-15: HOW THE INDRAG ATTRIBUTE IS USED	416
Table 15-1: HELPFUL THROWABLE METHODS	433
Table 16-1: GETTING THE CURRENT THREAD	447
Table 16-2: SLEEPING AND INTERRUPTING	447
Table 16-3: CHECKING THE INTERRUPTED STATUS	448
Table 16-4: THREAD CONSTRUCTORS	449
Table 16-5: STARTING A THREAD	449
Table 16-6: CALLING THE THREAD.yield() METHOD	455
Table 16-7: THREAD'S PRIORITY-RELATED METHODS	455
Table 16-8: THREADGROUP'S PRIORITY-RELATED METHODS	456
Table 16-9: THREAD'S join() METHODS	458
Table 16-10: OBJECT'S wait() AND notify() METHODS	470
Table 17-1: CORE COLLECTION INTERFACE CHARACTERISTICS	487

Table 17-2: NEW JAVA 5 CORE COLLECTION INTERFACES	500
Table 18-1: JAVA FILE I/O CLASSES BY CONSTRUCTOR ARGUMENT TYPE	511
Table 18-2: JAVA I/O CLASSES ORGANIZED BY FILE-TERMINAL, INTERMEDIATE, OR USER-FRONTING CHARACTERISTIC	512
Table 18-3: HANDY java.io CLASS COMBINATION REFERENCE	545
Table 20-1: CLIENT-SERVER PROJECT SPECIFICATION	588
Table 20-3: CLASS RESPONSIBILITY ASSIGNMENT	590
Table 20-2: CLIENT-SERVER PROJECT NOUN-VERB ANALYSIS	590
Table 20-4: FIRST ITERATION DESIGN CONSIDERATIONS AND DECISIONS	592
Table 20-5: SECOND ITERATION DESIGN CONSIDERATIONS AND DECISIONS	596
Table 20-6: THIRD ITERATION DESIGN CONSIDERATIONS AND DECISIONS	602
Table 20-7: THIRD ITERATION DESIGN CONSIDERATIONS AND DECISIONS	607
Table 20-8: FINAL ITERATION DESIGN CONSIDERATIONS AND DECISIONS	613
Table 21-1: HELPFUL MySQL MONITOR COMMANDS	646
Table 21-2: EMPLOYEE TRAINING MANAGEMENT SYSTEM CLASS DESCRIPTIONS	657
Table 22-1: INHERITANCE FORM DESCRIPTIONS	683
Table 23-1: OBJECT USAGE SCENARIO EVALUATION CHECKLIST	700
Table 23-2: APPLYING THE OBJECT USAGE SCENARIO EVALUATION CHECKLIST	701
Table 23-3: equals() METHOD EQUIVALENCE RELATION	703
Table 23-4: BLOCH'S equals() METHOD CRITERIA	704
Table 23-5: THE hashCode() GENERAL CONTRACT	706
Table 24-1: TERMS AND DEFINITIONS RELATED TO THE LSP	741
Table 25-1: PATTERN SPECIFICATION TEMPLATE	747
Table Appendix A-1: PROJECT APPROACH STRATEGY	781
Table Appendix A-2: DEVELOPMENT CYCLE	782
Table Appendix B-1: ASCII TABLE	783

List of Figures

FIGURE 1-1: ISOMORPHIC MAPPING BETWEEN PROBLEM DOMAIN AND DESIGN DOMAIN	8
FIGURE 1-2: RESULTS OF RUNNING EXAMPLE 1.2	16
FIGURE 1-3: JAVADOC TOOL BEING USED TO GENERATE TESTCLASS API DOCUMENTATION	20
FIGURE 1-4: EXAMPLE HTML DOCUMENTATION PAGE CREATED WITH JAVADOC	20
FIGURE 2-1: JAVA PROJECT CREATION PROCESS	28
FIGURE 2-2: SUN'S JAVA DOWNLOAD PAGE	32
FIGURE 2-3: CUSTOM SETUP DIALOG	32
FIGURE 2-4: WINDOW SHOWING SUBFOLDERS AND FILES IN THE j2sdk1.4.2_01 FOLDER	33
FIGURE 2-5: CONTENTS LISTING OF C:\j2sdk1.4.2_01\bin	33
FIGURE 2-6: SETTING THE PATH ENVIRONMENT VARIABLE IN MICROSOFT WINDOWS 2000/XP	34
FIGURE 2-7: TESTING PATH ENVIRONMENT VARIABLE BY TYPING javac	35
FIGURE 2-8: SETTING SOME IMPORTANT FOLDER OPTIONS	36
FIGURE 2-9: SAMPLECLASS PROJECT SOURCE-CODE DIRECTORY STRUCTURE	37
FIGURE 2-10: FINAL SAMPLECLASS SUBDIRECTORY STRUCTURE	37
FIGURE 2-11: COMPILING THE JAVA SOURCE FILES	38
FIGURE 2-12: SAMPLECLASS SUB-DIRECTORY STRUCTURE AFTER COMPILATION	38
FIGURE 2-13: THE .CLASS FILES ARE LOCATED IN THEIR PROPER PACKAGE STRUCTURE	38
FIGURE 2-14: RESULTS OF RUNNING APPLICATIONCLASS PROGRAM	39
FIGURE 2-15: JBuilder WITH NO OPEN PROJECTS	40
FIGURE 2-16: NEW PROJECT MENU	40
FIGURE 2-17: PROJECT WIZARD STEP 1 OF 3	41
FIGURE 2-18: PROJECT WIZARD STEP 2 OF 3	41
FIGURE 2-19: PROJECT WIZARD STEP 3 OF 3	42
FIGURE 2-20: CREATING APPLICATIONCLASS.JAVA SOURCE FILE	42
FIGURE 2-21: TESTPROJECT AFTER CREATING SAMPLECLASS.JAVA AND APPLICATIONCLASS.JAVA	43
FIGURE 2-22: PROJECT PROPERTIES DIALOG	44
FIGURE 2-23: RUNTIME CONFIGURATION PROPERTIES DIALOG	44
FIGURE 2-24: JBuilder PROJECT MENU SHOWING THE MAKE PROJECT ITEM	45
FIGURE 2-25: RUNNING TESTPROJECT	46
FIGURE 2-26: DOWNLOADING A LINUX JAVA SELF-EXTRACTING BIN FILE	47
FIGURE 2-27: EXECUTING THE SELF-EXTRACTING BIN FILE	47
FIGURE 2-28: NEW SUBDIRECTORY RESULTING FROM SDK EXTRACTION	47
FIGURE 2-29: CONTENTS OF THE j2sdk1.4.2_02 DIRECTORY	48
FIGURE 2-30: CHECKING ENVIRONMENT VARIABLES WITH THE ENV COMMAND	48
FIGURE 2-31: USING THE TREE COMMAND TO SHOW DIRECTORY STRUCTURE	50
FIGURE 2-32: RUNNING APPLICATIONCLASS IN THE LINUX ENVIRONMENT	50
FIGURE 2-33: Xcode NEW PROJECT ASSISTANT	51
FIGURE 2-34: NEW JAVA TOOL WINDOW	52
FIGURE 2-35: Xcode APPLICATIONCLASS PROJECT WINDOW	52
FIGURE 2-36: EDITING THE MANIFEST FILE TO REFLECT THE CORRECT PACKAGE LOCATION OF THE MAIN APPLICATION CLASS	53
FIGURE 2-37: RUNNING APPLICATIONCLASS PROJECT FROM Xcode IDE	53
FIGURE 2-38: EXECUTING THE MyApp.jar FILE USING THE java COMMAND	54
FIGURE 3-1: TIGHT SPIRAL DEVELOPMENT CYCLE DEPLOYMENT	61
FIGURE 3-2: ROBOT RAT VIEWED AS ATTRIBUTES	65
FIGURE 3-3: ROBOT RAT FLOOR SKETCH	65
FIGURE 3-4: COMPLETE ROBOT RAT ATTRIBUTES	66
FIGURE 3-5: ROBOTRAT UML CLASS DIAGRAM	68
FIGURE 3-6: COMPILING & TESTING ROBOTRAT CLASS - FIRST ITERATION	70
FIGURE 3-7: COMPILING & TESTING ROBOTRAT.CLASS - SECOND ITERATION	72
FIGURE 3-8: TESTING MENU COMMANDS	75

FIGURE 3-9: pen_position STATE TRANSITION DIAGRAM	76
FIGURE 3-10: STATE TRANSITION DIAGRAM FOR THE DIRECTION VARIABLE	77
FIGURE 3-11: printFloor() METHOD TEST	79
FIGURE 3-12: TESTING THE getSpaces() AND move() METHODS	82
FIGURE 3-13: TWO FLOOR PATTERNS PRINTED TO THE CONSOLE	82
FIGURE 3-14: PARTIAL ROBOTRAT JAVADOC DOCUMENTATION	83
FIGURE 4-1: TYPICAL POWER MAC G4 SYSTEM	92
FIGURE 4-2: SYSTEM UNIT	93
FIGURE 4-3: MAIN LOGIC BOARD BLOCK DIAGRAM	93
FIGURE 4-4: POWERPC G4 PROCESSOR	94
FIGURE 4-5: MOTOROLA POWERPC 7400 BLOCK DIAGRAM	94
FIGURE 4-6: MEMORY HIERARCHY	96
FIGURE 4-7: SIMPLIFIED MEMORY SUBSYSTEM DIAGRAM	96
FIGURE 4-8: SIMPLIFIED MAIN MEMORY DIAGRAM	97
FIGURE 4-9: PROCESSING CYCLE	99
FIGURE 4-10: DUMB SORT RESULTS 1	101
FIGURE 4-11: DUMB SORT RESULTS 2	101
FIGURE 4-12: DUMB SORT RESULTS 3	101
FIGURE 4-13: ALGORITHMIC GROWTH RATES	101
FIGURE 4-14: JAVA HOTSPOT™ VIRTUAL MACHINE TARGETS SPECIFIC HARDWARE PLATFORMS	102
FIGURE 4-15: JAVA HOTSPOT™ VIRTUAL MACHINE ARCHITECTURE	103
FIGURE 5-1: JAVA PLATFORM VERSION 1.4.2 PACKAGE ARCHITECTURAL OVERVIEW	110
FIGURE 5-2: JAVA 2 PLATFORM API VERSION 1.4.2 SPECIFICATION PAGE	111
FIGURE 5-3: PARTIAL LISTING FOR java.lang PACKAGE	111
FIGURE 5-4: DETAILED INFORMATION FOR java.lang.STRING CLASS	112
FIGURE 5-5: STRING INHERITANCE HIERARCHY UML DIAGRAM	113
FIGURE 5-6: JBUTTON INHERITANCE HIERARCHY INFORMATION	113
FIGURE 5-7: JBUTTON INHERITANCE HIERARCHY	114
FIGURE 5-8: JBUTTON INHERITED METHODS GROUPED BY BASE CLASS (PARTIAL LISTING)	115
FIGURE 5-9: setVisible() FUNCTION DESCRIPTION	115
FIGURE 6-1: COMPILING AND EXECUTING SimpleApplication	125
FIGURE 6-2: RESULTS OF RUNNING IdentifierTest PROGRAM	126
FIGURE 6-3: TestClassOne Mod 1 OUTPUT	131
FIGURE 6-4: TestClassOne Mod 3 OUTPUT	132
FIGURE 6-5: TestClassOne Mod 4 OUTPUT	132
FIGURE 6-6: TestClassOne Mod 5 OUTPUT	133
FIGURE 6-7: COMPILER ERROR MESSAGE RESULTING FROM ATTEMPT TO CHANGE A CONSTANT'S VALUE	133
FIGURE 6-8: RESULTS OF RUNNING TestClassOne WITH THE INPUT 1 2 3	134
FIGURE 6-9: RUNNING TestClassOne AGAIN WITH DIFFERENT INPUT VALUES	134
FIGURE 6-10: RESULTS OF RUNNING Example 6.12	135
FIGURE 6-11: CREATING AN OBJECT WITH THE NEW OPERATOR	136
FIGURE 6-12: CREATING ANOTHER OBJECT WITH THE NEW OPERATOR	136
FIGURE 6-13: REUSING THE object_reference VARIABLE	137
FIGURE 6-14: RESULTS OF RUNNING Example 6.14	138
FIGURE 6-15: RESULTS OF RUNNING Example 6.15	139
FIGURE 6-16: RESULTS OF RUNNING Example 6.16	142
FIGURE 6-17: RESULTS OF RUNNING Example 6.17	142
FIGURE 6-18: RESULTS OF RUNNING Example 6.18	143
FIGURE 6-19: RESULTS OF RUNNING Example 6.19	143
FIGURE 6-20: RESULTS OF RUNNING Example 6.20	144
FIGURE 6-21: RESULTS OF RUNNING Example 6.21	145
FIGURE 6-22: RESULTS OF RUNNING Example 6.22	145
FIGURE 6-23: RESULTS OF RUNNING Example 6.23	146
FIGURE 6-24: RESULTS OF RUNNING Example 6.24	146
FIGURE 6-25: RESULTS OF RUNNING Example 6.25	147
FIGURE 6-26: RESULTS OF RUNNING Example 6.26	148
FIGURE 6-27: BITWISE OPERATOR TRUTH TABLES	148
FIGURE 6-28: RESULTS OF RUNNING Example 6.27	148
FIGURE 6-29: RESULTS OF RUNNING Example 6.28	149

FIGURE 7-1: if STATEMENT EXECUTION DIAGRAM	156
FIGURE 7-2: Results of RUNNING Example 7.1	157
FIGURE 7-3: Results of RUNNING Example 7.2	157
FIGURE 7-4: Results of RUNNING Example 7.3	158
FIGURE 7-5: if/else STATEMENT EXECUTION DIAGRAM	158
FIGURE 7-6: Results of RUNNING Example 7.4	159
FIGURE 7-7: Results of RUNNING Example 7.5	159
FIGURE 7-8: switch STATEMENT EXECUTION DIAGRAM	160
FIGURE 7-9: Results of RUNNING Example 7.6	160
FIGURE 7-10: Results of RUNNING Example 7.7	161
FIGURE 7-11: Results of RUNNING Example 7.8	162
FIGURE 7-12: while STATEMENT EXECUTION DIAGRAM	163
FIGURE 7-13: Results of RUNNING Example 7.9	163
FIGURE 7-14: do/while STATEMENT EXECUTION DIAGRAM	164
FIGURE 7-15: Results of RUNNING Example 7-10	164
FIGURE 7-16: for STATEMENT EXECUTION DIAGRAM	165
FIGURE 7-17: Results of RUNNING Example 7.11	166
FIGURE 7-18: Results of RUNNING Example 7.12	166
FIGURE 7-19: Results of RUNNING CheckBookBALANCER	168
FIGURE 7-20: Results of RUNNING Example 7.14	169
FIGURE 7-21: Results of RUNNING Example 7.15	169
FIGURE 7-22: Results of RUNNING Example 7.16	170
FIGURE 7-23: Results of RUNNING Example 7.17 with Different Loop Limits	171
FIGURE 8-1: Array ELEMENTS ARE CONTIGUOUS AND HOMOGENEOUS	180
FIGURE 8-2: Specifying Array COMPONENT TYPE	181
FIGURE 8-3: Array-TYPE INHERITANCE Hierarchy	182
FIGURE 8-4: Results of RUNNING Example 8.1	184
FIGURE 8-5: MEMORY REPRESENTATION of PrIMITIVE Type Array int_ARRAY SHOWING Default Initialization	184
FIGURE 8-6: Results of RUNNING Example 8.2	185
FIGURE 8-7: ELEMENT VALUES of int_ARRAY After Initialization PERFORMED by SECOND for LOOP	185
FIGURE 8-8: Results of RUNNING Example 8.3	186
FIGURE 8-9: Results of RUNNING Example 8.4	186
FIGURE 8-10: Results of RUNNING Example 8.5	188
FIGURE 8-11: STATE of Affairs After Line 3 of Example 8.5 EXECUTES	188
FIGURE 8-12: STATE of Affairs After Line 5 of Example 8.5 EXECUTES.	189
FIGURE 8-13: STATE of Affairs After Line 10 of Example 8.5 EXECUTES	189
FIGURE 8-14: Final STATE of Affairs: All object_ARRAY ELEMENTS POINT TO AN Object object	190
FIGURE 8-15: Results of RUNNING Example 8.6	190
FIGURE 8-16: Results of RUNNING Example 8.7	191
FIGURE 8-17: Results of RUNNING Example 8.8	193
FIGURE 8-18: Array Declaration SYNTAX for a Two-Dimensional Array	194
FIGURE 8-19: A Two Dimensional Array with DIMENSIONS 10 by 10	195
FIGURE 8-20: Results of RUNNING Example 8.9	195
FIGURE 8-21: MEMORY REPRESENTATION of int_2d_ARRAY with 2 Rows AND 10 Columns	196
FIGURE 8-22: Results of RUNNING Example 8.10	197
FIGURE 8-23: Results of RUNNING Example 8.11	198
FIGURE 8-24: Results of RUNNING Example 8.12	200
FIGURE 8-25: Results of RUNNING Example 8.13	201
FIGURE 8-26: Results of RUNNING Example 8.14	201
FIGURE 9-1: PEOPLE MANAGEMENT PROGRAM PROJECT SPECIFICATION	211
FIGURE 9-2: CLASS DIAGRAM for PEOPLE MANAGER CLASSES	213
FIGURE 9-3: STATIC AND Non-STATIC FIELDS	215
FIGURE 9-4: HORIZONTAL Access CONTROLLED via Access Modifiers public AND private	216
FIGURE 9-5: METHOD DEFINITION STRUCTURE	218
FIGURE 9-6: Results of RUNNING Example 9.5	224
FIGURE 9-7: Results of RUNNING Example 9.7	225
FIGURE 9-8: Results of RUNNING Example 9.9	226
FIGURE 9-9: Results of RUNNING Example 9.11	227
FIGURE 9-10: Results of RUNNING Example 9.16	230

FIGURE 9-11: RESULTS OF RUNNING EXAMPLE 9.18	232
FIGURE 9-12: PRIMITIVE AND REFERENCE ARGUMENT VALUES ARE COPIED TO METHOD PARAMETERS	233
FIGURE 9-13: RESULTS OF RUNNING EXAMPLE 9.19	235
FIGURE 9-14: LINKED LIST WITH THREE NODES	239
FIGURE 10-1: UML DIAGRAM SHOWING SIMPLE AGGREGATION	247
FIGURE 10-2: PART CLASS SHARED BETWEEN SIMPLE AGGREGATE CLASSES	247
FIGURE 10-3: UML DIAGRAM SHOWING COMPOSITE AGGREGATION	247
FIGURE 10-4: SIMPLE AGGREGATION EXAMPLE	248
FIGURE 10-5: RESULTS OF RUNNING EXAMPLE 10.3	249
FIGURE 10-6: COMPOSITE AGGREGATION EXAMPLE	249
FIGURE 10-7: RESULTS OF RUNNING EXAMPLE 10.6	250
FIGURE 10-8: SEQUENCE DIAGRAM – SIMPLE AGGREGATION	250
FIGURE 10-9: SEQUENCE DIAGRAM – COMPOSITE AGGREGATION	251
FIGURE 10-10: AIRCRAFT ENGINE PROJECT SPECIFICATION	252
FIGURE 10-11: ENGINE SIMULATION CLASS DIAGRAM	253
FIGURE 10-12: ENGINE CLASS	253
FIGURE 10-13: AIRCRAFT ENGINE CREATE ENGINE OBJECT SEQUENCE	255
FIGURE 10-14: RESULT OF RUNNING EXAMPLE 10.8	256
FIGURE 10-15: SIMPLE AGGREGATION CLASS DIAGRAM	261
FIGURE 10-16: COMPOSITE AGGREGATION CLASS DIAGRAM	261
FIGURE 11-1: INHERITANCE HIERARCHY ILLUSTRATING GENERALIZED & SPECIALIZED BEHAVIOR	266
FIGURE 11-2: UML CLASS DIAGRAM SHOWING DERIVEDCLASS INHERITING FROM BASECLASS	268
FIGURE 11-3: UML DIAGRAM OF BASECLASS & DERIVEDCLASS SHOWING FIELDS AND METHODS	269
FIGURE 11-4: RESULTS OF RUNNING EXAMPLE 11.3	271
FIGURE 11-5: UML DIAGRAM SHOWING STUDENT CLASS INHERITANCE HIERARCHY	272
FIGURE 11-6: RESULTS OF RUNNING EXAMPLE 11.6	274
FIGURE 11-7: RESULTS OF RUNNING EXAMPLE 11.7	274
FIGURE 11-8: UML CLASS DIAGRAM FOR BASECLASS & DERIVEDCLASS	275
FIGURE 11-9: RESULTS OF RUNNING EXAMPLE 11.3 WITH MODIFIED VERSION OF DERIVEDCLASS	276
FIGURE 11-10: EXPRESSING AN ABSTRACT CLASS IN THE UML	277
FIGURE 11-11: UML CLASS DIAGRAM SHOWING THE ABSTRACTCLASS & DERIVEDCLASS INHERITANCE HIERARCHY	277
FIGURE 11-12: RESULTS OF RUNNING EXAMPLE 11.11	279
FIGURE 11-13: TWO TYPES OF UML INTERFACE DIAGRAMS	280
FIGURE 11-14: UML DIAGRAM SHOWING THE SIMPLE FORM OF REALIZATION	281
FIGURE 11-15: UML DIAGRAM SHOWING THE EXPANDED FORM OF REALIZATION	281
FIGURE 11-16: UML DIAGRAM SHOWING THE MESSAGEPRINTERCLASS REALIZING THE MESSAGEPRINTER INTERFACE	281
FIGURE 11-17: RESULTS OF RUNNING EXAMPLE 11.14	282
FIGURE 11-18: HORIZONTAL AND VERTICAL ACCESS IN MULTI-PACKAGE ENVIRONMENT	283
FIGURE 11-19: EMPLOYEE CLASS INHERITANCE HIERARCHY	286
FIGURE 11-20: RESULTS OF RUNNING EXAMPLE 11.24	288
FIGURE 11-21: UML CLASS DIAGRAM FOR AIRCRAFT ENGINE SIMULATOR	289
FIGURE 11-22: RESULTS OF RUNNING EXAMPLE 11.38	297
FIGURE 12-1: STANDARD ALGEBRAIC COORDINATE SYSTEM	307
FIGURE 12-2: STANDARD COMPUTER-SCREEN COORDINATE SYSTEM	307
FIGURE 12-3: COMPONENTS AND BOUNDS	308
FIGURE 12-4: TOP-LEVEL CONTAINER HIERARCHY	309
FIGURE 12-5: SCREEN SHOT OF AN EMPTY JWINDOW	309
FIGURE 12-6: STRUCTURE OF A JWINDOW	309
FIGURE 12-7: SCREENSHOT OF AN EMPTY JFRAME	310
FIGURE 12-8: STRUCTURE OF AN EMPTY JFRAME	310
FIGURE 12-9: JFRAME WITH MENUBAR	310
FIGURE 12-10: STRUCTURE OF JFRAME WITH MENUBAR	310
FIGURE 12-11: A JDialog WITH A LABEL AND THREE BUTTONS	310
FIGURE 12-12: TESTFRAME GUI	311
FIGURE 12-13: TESTFRAMEWITHCONTENTS GUI	317
FIGURE 12-14: TESTFRAMEWITHCONTENTS CONSOLE OUTPUT	317
FIGURE 12-15: TESTFRAMEWITHCONTENTS RESIZED LARGER	318
FIGURE 12-16: TESTFRAMEWITHCONTENTS RESIZED SMALLER	318
FIGURE 12-17: TESTFRAMEWITHFLOWLAYOUT GUI	319

FIGURE 12-18: TestFrameWithFlowLayout Console Output	319
FIGURE 12-19: TestFrameWithFlowLayout Resized Wider	319
FIGURE 12-20: TestFrameWithFlowLayout Resized Taller	319
FIGURE 12-21: COORDINATES FOR A SAMPLE GridLayout with 4 Rows and 2 Columns	320
FIGURE 12-22: TestFrameWithGridLayout GUI	321
FIGURE 12-23: TestFrameWithGridLayout Console Output	321
FIGURE 12-24: TestFrameWithGridLayout Resized Wider	321
FIGURE 12-25: TestFrameWithGridLayout Resized Taller	321
FIGURE 12-26: BorderLayout Positions	322
FIGURE 12-27: TestFrameWithBorderLayout GUI	322
FIGURE 12-28: TestFrameWithBorderLayout Console Output	323
FIGURE 12-29: TestFrameWithBorderLayout Resized Wider	323
FIGURE 12-30: TestFrameWithBorderLayout Resized Taller	323
FIGURE 12-31: GridBagLayoutExample GUI	324
FIGURE 12-32: GridBagLayoutExample Console Output	324
FIGURE 12-33: GridBagLayoutExample GUI Variation 1	325
FIGURE 12-34: GridBagLayoutExample GUI Variation 2	325
FIGURE 12-35: GridBagLayoutExample GUI Variation 3	325
FIGURE 12-36: GridBagLayoutExample GUI Variation 4	326
FIGURE 12-37: GridBagLayoutExample GUI Variation 5	326
FIGURE 12-38: CombinedLayoutsExample GUI	328
FIGURE 12-39: CombinedLayoutExample Console Output	329
FIGURE 12-40: JComponent Inheritance Hierarchy	329
FIGURE 12-41: MainFrame GUI	342
FIGURE 12-42: Visual Guide to the Components in MainFrame	343
FIGURE 12-43: MainFrame Layout	344
FIGURE 12-44: Exercise3: Default Size	345
FIGURE 12-45: Exercise3: Stretched Horizontally	346
FIGURE 12-46: Alternate Border Layout	346
FIGURE 13-1: EventObject Inheritance Hierarchy	350
FIGURE 13-2: ACME Product Services Confirmation	353
FIGURE 13-3: Event-Handling Division of Labor	353
FIGURE 13-4: Inheritance Hierarchy for the Examples Used in this Chapter	354
FIGURE 13-5: ActionEvent Inheritance Hierarchy	356
FIGURE 13-6: ActionListener Inheritance Hierarchy	357
FIGURE 13-7: MouseEvent Inheritance Hierarchy	359
FIGURE 13-8: MouseListener, MouseMotionListener, and MouseWheelListener Inheritance Hierarchy	361
FIGURE 13-9: KeyEvent Inheritance Hierarchy	364
FIGURE 13-10: KeyListener Inheritance Hierarchy	365
FIGURE 13-11: ChangeEvent Inheritance Hierarchy	371
FIGURE 13-12: ChangeListener Inheritance Hierarchy	372
FIGURE 13-13: ListSelectionEvent Inheritance Hierarchy	373
FIGURE 13-14: ListSelectionListener Inheritance Hierarchy	373
FIGURE 14-1: This Chapter's Completed Application	381
FIGURE 14-2: Graphics Drawing Operations and Property-Related Methods	383
FIGURE 14-3: A JFrame Containing a JList with a Custom ListCellRenderer	394
FIGURE 14-4: A JFrame Containing a JTable	394
FIGURE 14-5: A JFrame Containing a Highly Customized JTree	395
FIGURE 14-6: Sequence Diagram for a JList Using CheckboxListCell – First Version	399
FIGURE 14-7: Maneuvering Through the Swing API	400
FIGURE 14-8: Sequence Diagram for a JList Using CheckboxListCell – Second Version	408
FIGURE 14-9: TableCellEditor and TableCellEditor Inheritance Hierarchy	409
FIGURE 14-10: DefaultListModel Inheritance Hierarchy	414
FIGURE 15-1: Throwable Class Hierarchy	428
FIGURE 15-2: NumberFormatException Class Inheritance Hierarchy	429
FIGURE 15-3: Results of Running Example 15.1 with Good and Bad Input Strings	430
FIGURE 15-4: Results of Running Example 15.2	431
FIGURE 15-5: Results of Running Example 15.5	434
FIGURE 15-6: Results of Running Example 15.6	435

FIGURE 15-7: RESULTS OF RUNNING EXAMPLE 15.7	476
FIGURE 15-8: RESULTS OF RUNNING EXAMPLE 15.9	478
FIGURE 15-9: RESULTS OF RUNNING EXAMPLE 15.11	479
FIGURE 16-1: RESULTS OF RUNNING EXAMPLE 16.1	445
FIGURE 16-2: RESULTS OF RUNNING EXAMPLE 16.2	446
FIGURE 16-3: RESULTS OF RUNNING EXAMPLE 16.11	458
FIGURE 16-4: RESULTS OF RUNNING EXAMPLE 16.13	459
FIGURE 16-5: BREAKER.JAVA THREAD INTERACTION	460
FIGURE 16-6: ACQUIRING AND RELEASING LOCKS	461
FIGURE 16-7: BREAKER(2) AND BREAKER(3) BOTH SUCCEED	463
FIGURE 16-8: BREAKER(2) FAILS BECAUSE NOT ALL THREADS SYNCHRONIZED	464
FIGURE 16-9: BREAKER(3) FAILS BECAUSE NOT ALL THREADS SYNCHRONIZED	464
FIGURE 16-10: RESULTS OF RUNNING EXAMPLE 16.15	465
FIGURE 16-11: BREAKER(2) FAILS BECAUSE THREADS SYNCHRONIZED ON DIFFERENT LOCKS	465
FIGURE 16-12: RESULTS OF RUNNING EXAMPLE 16.16	466
FIGURE 16-13: CONSUMER THREAD WAITS	472
FIGURE 16-14: PRODUCER THREAD WAITS	473
FIGURE 16-15: DEADLOCKED THREADS	474
FIGURE 16-16: RESULTS OF RUNNING EXAMPLE 16.25	476
FIGURE 16-17: DEADLOCK DUE TO NESTED SYNCHRONIZATION	476
FIGURE 17-1: RESULTS OF TESTING DYNAMICARRAY	483
FIGURE 17-2: RESULTS OF RUNNING EXAMPLE 17.3	485
FIGURE 17-3: RESULTS OF RUNNING EXAMPLE 17.4	485
FIGURE 17-4: JAVA 1.4.2 COLLECTIONS FRAMEWORK CORE INTERFACE HIERARCHY	487
FIGURE 17-5: ARRAY OF OBJECT REFERENCES BEFORE INSERTION	488
FIGURE 17-6: NEW REFERENCE TO BE INSERTED AT ARRAY ELEMENT 3 (INDEX 2)	488
FIGURE 17-7: ARRAY AFTER NEW REFERENCE INSERTION	489
FIGURE 17-8: LINKED LIST NODE ORGANIZATION	489
FIGURE 17-9: LINKED LIST BEFORE NEW ELEMENT INSERTION	489
FIGURE 17-10: NEW REFERENCE BEING INSERTED INTO SECOND ELEMENT POSITION	490
FIGURE 17-11: REFERENCES OF PREVIOUS, NEW, AND NEXT LIST ELEMENTS MUST BE MANIPULATED	490
FIGURE 17-12: LINKED LIST INSERTION COMPLETE	490
FIGURE 17-13: A HASH FUNCTION TRANSFORMS A KEY VALUE INTO AN ARRAY INDEX	491
FIGURE 17-14: HASH TABLE COLLISIONS ARE RESOLVED BY LINKING NODES TOGETHER	491
FIGURE 17-15: RED-BLACK TREE NODE DATA ELEMENTS	492
FIGURE 17-16: RED-BLACK TREE AFTER INSERTING INTEGER VALUES 9, 3, 5, 6, 7, 8, 4, 1	492
FIGURE 17-17: RESULTS OF RUNNING EXAMPLE 17.5	494
FIGURE 17-18: RESULTS OF RUNNING EXAMPLE 17.8	496
FIGURE 17-19: RESULTS OF RUNNING EXAMPLE 17.9	497
FIGURE 17-20: RESULTS OF RUNNING EXAMPLE 17.12	499
FIGURE 17-21: JAVA 5 COLLECTIONS FRAMEWORK CORE INTERFACE HIERARCHY	500
FIGURE 17-22: RESULTS OF RUNNING EXAMPLE 17.13	501
FIGURE 17-23: RESULTS OF RUNNING EXAMPLE 17.14	502
FIGURE 18-1: PARTIAL JAVA.IO PACKAGE HIERARCHY	509
FIGURE 18-2: RESULTS OF RUNNING EXAMPLE 18.1	513
FIGURE 18-3: RESULTS OF RUNNING EXAMPLE 18.2	514
FIGURE 18-4: RESULTS OF RUNNING EXAMPLE 18.3	516
FIGURE 18-5: CONTENTS OF TEST.TXT AFTER EXECUTING EXAMPLE 18.3 SIX TIMES	516
FIGURE 18-6: RESULTS OF RUNNING EXAMPLE 18.4	517
FIGURE 18-7: CONTENTS OF TEST.TXT FILE AFTER EXECUTING EXAMPLE 18.4	517
FIGURE 18-8: RESULTS OF RUNNING EXAMPLE 18.5	518
FIGURE 18-9: CONTENTS OF TEST.TXT AFTER RUNNING EXAMPLE 18.5	518
FIGURE 18-10: RESULTS OF RUNNING EXAMPLE 18.7	519
FIGURE 18-11: CONTENTS OF PEOPLE.DAT FILE VIEWED WITH TEXT EDITOR	520
FIGURE 18-12: CONTENTS OF OUTPUT.TXT FILE AFTER EXAMPLE 18.8 EXECUTES	520
FIGURE 18-13: RESULTS OF RUNNING EXAMPLE 18.9	521
FIGURE 18-14: RESULTS OF RUNNING EXAMPLE 18.10	522
FIGURE 18-15: RESULTS OF RUNNING EXAMPLE 18.11	523
FIGURE 18-16: WARNING PRODUCED WHEN COMPILING EXAMPLE 18.12	524

FIGURE 18-17: RESULTS OF RUNNING EXAMPLE 18.12	524
FIGURE 18-18: CONTENTS OF TEST.TXT FILE AFTER EXAMPLE 18.13 EXECUTES	525
FIGURE 18-19: CONTENTS OF TEST.TXT AFTER EXAMPLE 18.14 EXECUTES	526
FIGURE 18-20: CONTENTS OF TEST.TXT FILE AFTER EXAMPLE 18.15 EXECUTES	526
FIGURE 18-21: CONTENTS OF TEST.TXT FILE AFTER EXAMPLE 18.16 EXECUTES	527
FIGURE 18-22: RESULTS OF RUNNING EXAMPLE 18.17	528
FIGURE 18-23: RESULTS OF RUNNING EXAMPLE 18.18	528
FIGURE 18-24: RESULTS OF RUNNING EXAMPLE 18.19	529
FIGURE 18-25: INITIAL EXECUTION OF PROPERTIESTESTERAPP (EXAMPLE 18.21)	530
FIGURE 18-26: CONTENTS OF APP_PROP.XML AFTER EXAMPLE 18.21 EXECUTES	531
FIGURE 18-27: LEGACY DATA FILE ADAPTER PROJECT SPECIFICATION	532
FIGURE 18-28: CONTENTS OF BOOKS.DAT EXAMPLE LEGACY DATAFILE VIEWED WITH TEXT EDITOR	533
FIGURE 18-29: HEADER AND RECORD LENGTH ANALYSIS	533
FIGURE 18-30: RESULTS OF RUNNING EXAMPLE 18.29	544
FIGURE 19-1: A SIMPLE COMPUTER NETWORK	554
FIGURE 19-2: LOCAL AREA NETWORK CONNECTED TO THE INTERNET	555
FIGURE 19-3: THE INTERNET – A NETWORK OF NETWORKS COMMUNICATING VIA INTERNET PROTOCOLS	556
FIGURE 19-4: CLIENT AND SERVER HARDWARE AND APPLICATIONS	557
FIGURE 19-5: CLIENT AND SERVER APPLICATIONS PHYSICALLY DEPLOYED TO SAME COMPUTER	558
FIGURE 19-6: CLIENT AND SERVER APPLICATIONS REQUIRE SEPARATE JAVA VIRTUAL MACHINES	559
FIGURE 19-7: STARTING MULTIPLE TERMINAL WINDOWS USING START COMMAND	559
FIGURE 19-8: MULTIPLE JVMs LAUNCHED AS SEPARATE PROCESSES IN MAC OSX	560
FIGURE 19-9: KILLING UNIX PROCESSES WITH THE KILL COMMAND	560
FIGURE 19-10: RUNNING MULTIPLE CLIENT JVMs ON SAME HARDWARE	561
FIGURE 19-11: CLIENT AND SERVER APPLICATIONS DEPLOYED ON DIFFERENT COMPUTERS	561
FIGURE 19-12: PHYSICALLY DISTRIBUTED CLIENT AND SERVER APPLICATIONS NEED A JVM	562
FIGURE 19-13: A MULTI-TIERED APPLICATION	562
FIGURE 19-14: PHYSICALLY DEPLOYING LOGICAL APPLICATION TIERS ON SAME COMPUTER	563
FIGURE 19-15: LOGICAL APPLICATION TIERS PHYSICALLY DEPLOYED TO DIFFERENT COMPUTERS	563
FIGURE 19-16: TCP/IP PROTOCOL STACK	564
FIGURE 19-17: INTERNET PROTOCOL STACK OPERATIONS	566
FIGURE 19-18: JAVA SERVER APPLICATION UTILIZING A SERVERSOCKET OBJECT	567
FIGURE 19-19: INCOMING CLIENT CONNECTION	568
FIGURE 19-20: CONNECTION BETWEEN CLIENT & SERVER ESTABLISHED	568
FIGURE 19-21: RETRIEVE IOSTREAM OBJECTS FROM SERVER AND CLIENT SOCKET OBJECTS	568
FIGURE 19-22: RESULTS OF RUNNING EXAMPLE 19.1	569
FIGURE 19-23: THE REMOTE METHOD INVOCATION (RMI) CONCEPT	570
FIGURE 19-24: CLASS DIAGRAM FOR REMOTESYSTEMMONITORINTERFACE & REMOTESYSTEMMONITORIMPLEMENTATION	572
FIGURE 19-25: SYSTEMMONITORSERVER RUNNING ON HOST MACHINE	574
FIGURE 19-26: RESULTS OF RUNNING THE SYSTEMMONITORCLIENT APPLICATION CONNECTING TO THE LOCALLY SERVED SERVER APPLICATION	574
FIGURE 19-27: RESULTS OF THE SYSTEMMONITORCLIENT APPLICATION AFTER RUNNING ON A REMOTE PC	574
FIGURE 19-28: SYSTEMMONITORCLIENT INVOKING THE REMOTE METHOD ON A PC RUNNING THE SYSTEMMONITORSERVER APPLICATION	575
FIGURE 20-1: CLIENT AND SERVER APPLICATIONS	582
FIGURE 20-2: INCOMING CLIENT CONNECTION	583
FIGURE 20-3: THE CONNECTION IS ESTABLISHED – THERE ARE SOCKETS AT BOTH ENDS OF THE CONNECTION	583
FIGURE 20-4: THE SOCKET OBJECTS ARE USED TO RETRIEVE THE IOSTREAM OBJECTS	583
FIGURE 20-5: SIMPLESERVER RUNNING & WAITING FOR INCOMING CLIENT CONNECTIONS	586
FIGURE 20-6: SIMPLECLIENT CONSOLE OUTPUT AND GUI	587
FIGURE 20-7: SIMPLESERVER CONSOLE AFTER DETECTING INCOMING CLIENT CONNECTION	587
FIGURE 20-8: SEVERAL MESSAGES EXCHANGED WITH THE SERVER FROM SIMPLECLIENT	587
FIGURE 20-9: RAT.gif	589
FIGURE 20-10: FIRST DRAFT CLASS DIAGRAM FOR THE NETRATSERVER APPLICATION	591
FIGURE 20-11: NETRATSERVER APPLICATION UPON START-UP	596
FIGURE 20-12: NETRATSERVER APPLICATION AFTER APPROXIMATELY 10 MOVE BUTTON CLICKS	596
FIGURE 20-13: RMI-ENABLED NETRATSERVER APPLICATION AT START-UP	599
FIGURE 20-14: RMI-ENABLED NETRATSERVER APPLICATION AFTER APPROXIMATELY 10 MOVE BUTTON CLICKS	599
FIGURE 20-15: RMI_NETRATCLIENT APPLICATION	601
FIGURE 20-16: THE FLOOR AFTER TESTING RMI_NETRATCLIENT	601
FIGURE 20-17: UPDATED ROBOT RAT SERVER APPLICATION CLASS DIAGRAM	603

FIGURE 20-18: THE FLOOR AFTER APPROXIMATELY 15 CLICKS OF THE SERVER-SIDE MOVE BUTTON	606
FIGURE 20-19: SERVER FLOOR AFTER RMI-CLIENT-CONTROLLED ROBOT RAT MOVES SOUTH SEVERAL CLICKS	607
FIGURE 20-20: SERVER FLOOR AFTER SECOND RMI-CLIENT-CONTROLLED ROBOT RAT APPEARS	607
FIGURE 20-21: FINAL NETRATSERVER APPLICATION DESIGN CLASS DIAGRAM	614
FIGURE 20-22: CONSOLE OUTPUT ON NETRATSERVER APPLICATION STARTUP	619
FIGURE 20-23: EMPTY FLOOR DISPLAYED AS A RESULT OF EXPLICITLY LOADING THE ROBOTRAT CLASS	619
FIGURE 21-1: BASICAPPLET INHERITANCE HIERARCHY	627
FIGURE 21-2: BASICAPPLET RUNNING IN WEB BROWSER	628
FIGURE 21-3: CONSOLE LOG SHOWING BASICAPPLET LIFE CYCLE MESSAGES	629
FIGURE 21-4: CONSOLE LOG SHOWING BASICAPPLET LIFE CYCLE MESSAGES AFTER BROWSER SHUTS DOWN	629
FIGURE 21-5: APPLETT LIFE CYCLE STAGES	629
FIGURE 21-6: THE <applet> TAG AND ITS ATTRIBUTES	630
FIGURE 21-7: AN APPLETT CAN ONLY CONNECT TO THE SERVER FROM WHICH IT WAS SERVED	631
FIGURE 21-8: APPLETTSERVER APPLETT RUNNING IN A BROWSER AND BEING ACCESSED BY THE SIMPLECLIENT APPLICATION	633
FIGURE 21-9: RESULTS OF ATTEMPTING TO CONNECT TO APPLETTSERVER FROM A COMPUTER OTHER THAN ITS SERVER	634
FIGURE 21-10: RESULTS OF RUNNING PARAMETERAPPLETT	636
FIGURE 21-11: POETRY APPLETT IN ACTION	639
FIGURE 21-12: EMPLOYEE TRAINING MANAGEMENT SYSTEM ARCHITECTURE DIAGRAM	642
FIGURE 21-13: SERVER-SIDE COMPONENT CLASS DIAGRAM	642
FIGURE 21-14: CLIENT-SIDE COMPONENT CLASS DIAGRAM	642
FIGURE 21-15: MYSQL MONITOR PROGRAM ON STARTUP	645
FIGURE 21-16: RESULTS OF ENTERING "SHOW DATABASES;" AT THE MONITOR PROMPT	646
FIGURE 21-17: RESULTS OF CHANGING TO THE MYSQL DATABASE WITH "USE MYSQL;" AND ENTERING "SHOW TABLES;"	646
FIGURE 21-18: STRUCTURE OF THE USER TABLE LOCATED IN THE MYSQL DATABASE	647
FIGURE 21-19: STRUCTURE OF THE DB TABLE	648
FIGURE 21-20: CONTENTS OF THE DB TABLE	648
FIGURE 21-21: STRUCTURE OF THE TABLES_PRIV TABLE	649
FIGURE 21-22: STRUCTURE OF THE COLUMNS_PRIV TABLE	649
FIGURE 21-23: STRUCTURE OF THE HOST TABLE	649
FIGURE 21-24: ENTITY DIAGRAM FOR EMPLOYEE AND EMPLOYEE_TRAINING TABLES	650
FIGURE 21-25: NEWLY CREATED CHAPTER_21 DATABASE TABLES	651
FIGURE 21-26: RESULTS OF EXECUTING THE SELECT STATEMENT AGAINST THE EMPLOYEES TABLE	652
FIGURE 21-27: RESULTS OF SELECTING ONLY THE FIRST_NAME AND LAST_NAME COLUMNS FROM THE EMPLOYEES TABLE	652
FIGURE 21-28: RESULTS OF THE UPDATE STATEMENT – NOTE THE MIDDLE_NAME IS CHANGED TO 'W'	653
FIGURE 21-29: EMPLOYEES TABLE WITH ADDITIONAL DATA ADDED	653
FIGURE 21-30: EMPLOYEE_TRAINING TABLE POPULATED WITH DATA	653
FIGURE 21-31: RESULTS OF JOINING THE EMPLOYEES TABLE WITH THE EMPLOYEE_TRAINING TABLE	653
FIGURE 21-32: RESULTS OF EXECUTING THE NESTED SELECT STATEMENT SHOWN IN EXAMPLE 21.16	654
FIGURE 21-33: RESULTS OF RUNNING EXAMPLE 21.17	655
FIGURE 21-34: RESULTS OF RUNNING EXAMPLE 21.18 WITH EMPLOYEE TABLE METADATA DISPLAYED	656
FIGURE 21-35: EMPLOYEE TRAINING MANAGEMENT SYSTEM ARCHITECTURE	656
FIGURE 21-36: EMPLOYEE TRAINING MANAGEMENT SYSTEM SOURCE CODE PACKAGE STRUCTURE	657
FIGURE 21-37: TERMINAL OUTPUT SHOWING DBSERVERAPP STARTUP SEQUENCE	668
FIGURE 21-38: EMPLOYEETRAININGAPPLETT APPEARANCE ON FIRST ACCESS	669
FIGURE 21-39: COMPLETE LIST OF EMPLOYEES	669
FIGURE 21-40: TRAINING RECORDS FOR HOMER SIMPSON	669
FIGURE 21-41: ADD NEW EMPLOYEE DIALOG	670
FIGURE 21-42: ADD NEW EMPLOYEE DIALOG WITH TEXT FIELDS FILLED IN	670
FIGURE 21-43: NEW EMPLOYEE ADDED TO THE DATABASE	670
FIGURE 22-1: MEYER'S INHERITANCE TAXONOMY	683
FIGURE 22-2: PERSON-EMPLOYEE INHERITANCE DIAGRAM	685
FIGURE 22-3: REVISED PERSON - EMPLOYEE EXAMPLE	689
FIGURE 22-4: RESULTS OF RUNNING EXAMPLE 22.9	693
FIGURE 23-1: RESULTS OF RUNNING EXAMPLE 23.4	705
FIGURE 23-2: RESULTS OF RUNNING EXAMPLE 23.6	707
FIGURE 23-3: CONCEPT OF A SHALLOW COPY	708
FIGURE 23-4: CONCEPT OF A DEEP COPY	709
FIGURE 23-5: RESULTS OF RUNNING EXAMPLE 23.8	710
FIGURE 23-6: RESULTS OF RUNNING EXAMPLE 23.10	712

FIGURE 23-7: RESULTS OF RUNNING EXAMPLE 23.11	714
FIGURE 23-8: RESULTS OF RUNNING EXAMPLE 23.13	715
FIGURE 24-1: RESULTS OF RUNNING EXAMPLE 24.2	726
FIGURE 24-2: RESULTS OF RUNNING EXAMPLE 24.4	728
FIGURE 24-3: RESULTS OF RUNNING EXAMPLE 24.6	730
FIGURE 24-4: RESULTS OF RUNNING EXAMPLE 24.8	732
FIGURE 24-5: STRONG VS. WEAK TYPES	733
FIGURE 24-6: RESULTS OF RUNNING EXAMPLE 24.12	734
FIGURE 24-7: NAVAL FLEET CLASS INHERITANCE HIERARCHY	737
FIGURE 24-8: RESULTS OF RUNNING EXAMPLE 24.22	740
FIGURE 24-9: TRADITIONAL TOP-DOWN FUNCTIONAL DEPENDENCIES	740
FIGURE 25-1: RESULTS OF RUNNING EXAMPLE 25.8	751
FIGURE 25-2: MODEL-VIEW-CONTROLLER PATTERN	752
FIGURE 25-3: RESULTS OF RUNNING EXAMPLE 25.11 AND CLICKING THE "NEXT MESSAGE" BUTTON SEVERAL TIMES	753
FIGURE 25-4: EMPLOYEE MANAGEMENT APPLICATION UML CLASS DIAGRAM	757
FIGURE 25-5: INTERACTING WITH THE EMPLOYEE MANAGEMENT APPLICATION	776

PREFACE

WELCOME – AND THANK YOU!

Welcome to *Java For Artists: The Art, Philosophy, and Science of Object-Oriented Programming*. If you purchased this book for personal use Raffi and I would like to say thank-you for supporting our writing efforts. If you are standing in the bookstore reading this paragraph the following message is meant especially for you: Due to the economics of small print runs this book is more expensive than similar-sized books sitting on the same shelf. However, we invite you to compare carefully the contents and coverage of competing books before shelling out your hard-earned cash. We truly believe you'll agree with us that this book gives you much more value for your money. If you were forced to buy this book at full price from a college bookstore rest assured that it's one of the few text books you'll actually want to keep after you've finished the class. However you came upon this book we're glad you're here and we sincerely hope the time you spend reading it is time well spent.

TARGET AUDIENCE

Java For Artists targets both the undergraduate computer science or information technology student and the practicing programmer. It is both an introductory-level textbook and trade book.

As a textbook it employs learning objectives, skill-building exercises, suggested projects, and self-test questions to reinforce the learning experience. The projects offered range from the easy to the extremely challenging. It covers all the topics you'd expect to find in an introductory Java programming textbook and then some.

As a trade book it goes purposefully deeper into topics cut short or avoided completely in most introductory textbooks. Its coverage of advanced GUI programming techniques, network programming and object-oriented theory will enable you to take your skills to a higher level.

APPROACH(ES)

The Java platform, now in version 1.5 (Java 5) is so complex that any text or trade book that attempts a broad coverage of the topic must employ a multitude of approaches. The many approaches employed in *Java For Artists* include the following:

Say what we're gonna say; say it; then say what we said: This approach is supported by the copious use of chapter learning objectives, chapter introductions, quick review sections, and comprehensive summaries.

Test to the learning objectives: The material presented in each chapter is reinforced by end-of-chapter skill-building exercises, suggested projects, and self-test questions.

Repeat, repeat, repeat: If we think something is especially important we will present it to you in several different ways in different chapters in the book.

Illustrate, illustrate, illustrate: Pictures are worth a thousand words. To this end we tried to illustrate difficult concepts graphically where possible.

Demonstrate, demonstrate, demonstrate: The results of running almost every programming example in this book are shown via a screen or console capture. The relationship between what you see in the code and what you see as a result of running the code is clearly explained.

Every programming example must work as advertised: Nothing is more frustrating to a beginner than to enter a source code example from a book and try to compile it only to find that it doesn't work. All source code in this book is compiled and tested repeatedly before being cut and pasted into the page layout software. Line numbers are automatically added so humans don't mess things up.

Show real-world examples: The later programming examples in this book are more complex than any provided in any competing Java text book. This approach is necessary to push you past the stage of simplistic Java programming.

Show complete examples: At the risk of having too much source code we chose to provide complete examples to preserve the context of the code under discussion.

Include coverage of Java 1.4 and Java 5 programming techniques: Although the Java platform has evolved, students should be still be exposed to Java 1.4 programming techniques for the simple fact that they may find themselves maintaining legacy Java code. It's too early for textbooks to focus exclusively on Java 5. A student who understands Java 1.4 programming techniques will find it easier to make the move to Java 5 rather than the other way around.

Offer an advanced treatment of object-oriented design principles: Students are better prepared to tackle complex projects when they've been introduced to advanced object-oriented programming concepts. If they try and design without it they are shooting from the hip.

ORGANIZATION

Java For Artists is organized into six parts containing 25 chapters. Each part and the contents of its accompanying chapters is discussed below.

PART I: THE JAVA STUDENT SURVIVAL GUIDE

The four chapters in part I provide important background information meant to help the reader approach the material presented in the later sections of the book. Some of the material discussed in part I, especially in chapter 3, may be too advanced to be fully understood by the novice upon first reading.

CHAPTER 1 - AN APPROACH TO THE ART OF PROGRAMMING

Chapter 1 presents a general discussion of the practical aspects of Java programming. It starts by highlighting the difficulties students will encounter when first learning Java and discusses the emotions they will feel when they begin to experience the overwhelming complexity involved with learning the syntax of the Java language, the Java API, a suitable development environment, and how to solve a problem with a computer. The chapter also presents and discusses the three software development roles students play and the activities required by each role. It goes on to discuss a general approach to the art of programming which includes a treatment on the concept of the flow, which is a transcendental state programmers and artists attain when completely absorbed in their work. The chapter concludes with a discussion of how to manage Java project physical and conceptual complexity.

CHAPTER 2 - SMALL VICTORIES

Chapter 2 shows the reader how to create, compile, and run a simple Java project in several different development environments on Microsoft Windows, Apple's OS X, and Linux operating systems. It shows the reader how to download, install, and configure the Java platform for Microsoft Windows and Linux including a discussion on how to set the PATH and CLASSPATH environment variables. The chapter concludes with a discussion on how to create executable jar files.

CHAPTER 3 - PROJECT WALKTHROUGH: A COMPLETE EXAMPLE

Chapter 3 presents the execution of a complete Java programming project from specification to final testing. The project approach strategy and development cycle originally presented in chapter 1 are applied in a practical setting. Each iteration of the project demonstrates the use of the development cycle to include planning, coding, testing, and integration. The chapter's overarching theme is the answer to this question: *"How do you, as a beginner, maintain a sense of forward momentum on a programming project when you don't yet have a complete grasp of the Java language, API, or your development environment?"*

CHAPTER 4 - COMPUTERS, PROGRAMS, AND ALGORITHMS

Chapter 4 provides background information on how computers and programs work. It discusses the basics of computer hardware organization down to the processor level and the fundamentals of computer memory organization. It discusses the concept of a program and defines the steps of the processing cycle. The chapter concludes with a detailed discussion of the Java virtual machine (JVM) and its architecture.

PART II: LANGUAGE FUNDAMENTALS

Part II contains chapters 5 through 11 and presents the core Java language features to include an overview of the Java API, language syntax, operators, control-flow statements, arrays, classes, compositional design and design by inheritance.

CHAPTER 5 - OVERVIEW OF THE JAVA API

Chapter 5 presents an overview of the Java platform application programming interface (API). The chapter shows readers how to consult the Java online documentation to find out information about a Java API class they want to use and how to navigate a class inheritance hierarchy to discover the class's full functionality.

CHAPTER 6 - SIMPLE JAVA PROGRAMS: USING PRIMITIVE AND REFERENCE DATA TYPES

Chapter 6 shows the reader how to write simple Java programs using primitive and reference data types. It starts by providing a definition of a Java program and discusses the difference between an application object vs. an applet object. The chapter then goes on to demonstrate how to create and compile a simple Java application. Next, the chapter discusses the concept of identifiers and identifier naming rules. It presents and discusses the Java reserved keywords, type categories, and language operators. It illustrates how to work with primitive and reference types, defines the terms variable, constant, statement, and expression, and demonstrates the accessibility between class and instance fields and the main() method. In summary, this chapter provides the fundamental footing all readers must have before attempting the remaining chapters in the book.

CHAPTER 7 - CONTROLLING THE FLOW OF PROGRAM EXECUTION

Chapter 7 presents a complete treatment of Java's control flow statements to include the if, if/else, switch, while, do/while, and for. The chapter shows how to achieve complex program flow control by combining various control-flow statements via chaining and nesting. The chapter also presents the purpose and use of the break and continue statements.

CHAPTER 8 - ARRAYS

Chapter 8 offers a detailed discussion on the topic of Java arrays. The chapter begins by highlighting the special properties of Java array-type objects. The discussion then focuses on the functionality provided by single-dimensional arrays, followed, naturally, by multi-dimensional arrays. The material is reinforced with many illustrations showing readers how Java arrays are organized in memory. Tons of programming examples serve to reinforce the reader's understanding of array concepts.

CHAPTER 9 - TOWARD PROBLEM ABSTRACTION: CREATING NEW DATA TYPES

Chapter 9 introduces the reader formally to the Java class construct. It starts with a discussion of problem abstraction and where the process of problem abstraction fits into the development cycle. The concept of the Unified Modeling Language (UML) class diagram is presented and is used from this point forward to communicate program designs. The Java class construct is thoroughly treated including class members, access modifiers, and static and non-static fields and methods. The class method is treated in detail as well and includes a discussion of method modifiers, return types, method names, parameter lists, and the method body. Numerous programming examples reinforce chapter material. The chapter concludes with a discussion of static initializers.

CHAPTER 10 - COMPOSITIONAL DESIGN

Chapter 10 shows readers how to create new class types from existing class types using compositional design. The chapter discusses the concepts of dependency vs. association and simple aggregation vs. composite aggregation. The chapter expands the reader's knowledge of UML by showing how to express various types of aggregation via a UML class diagram. The chapter also introduces readers to the UML sequence diagram in conjunction with the chapter's primary programming example. Other topics discussed in the chapter include the typesafe enumeration pattern and how to compile multiple Java source files with the javac command-line tool.

CHAPTER 11 - EXTENDING CLASS BEHAVIOR THROUGH INHERITANCE

Chapter 11 shows readers how to create new Java classes using inheritance. The chapter starts by showing how to express inheritance relationships via a UML class diagram. It then moves on to discuss the three essential purposes of inheritance, the concepts of base and derived classes, and how to override base class methods in derived classes. Next, the chapter presents a discussion of abstract base classes, abstract methods, and interfaces. It highlights the differences between abstract base classes vs. interfaces and shows readers how to represent interfaces in a UML class diagram. The chapter also presents a discussion of how to control horizontal and vertical access, and how to achieve polymorphic behavior. The primary chapter programming example shows readers how to combine compositional design with inheritance to achieve powerful results.

PART III: GRAPHICAL USER INTERFACE PROGRAMMING

Part III includes chapters 12 through 14 and gives readers a detailed treatment of graphical user interface (GUI) programming using AWT and Swing components.

CHAPTER 12 - JAVA SWING API OVERVIEW

Chapter 12 presents the fundamental concepts of GUI programming including an overview of the fundamental components found in the Java Swing API. This chapter introduces readers to the screen coordinate system and explains the different types of windows. It discusses the differences between containers and components and explains the use and behavior of layout managers.

CHAPTER 13 - HANDLING GUI EVENTS

Chapter 13 moves beyond chapter 12 and shows readers how to create rich, interactive GUIs utilizing Java event handling techniques. It dives deeply into the different types of `EventListener` interfaces and describes the steps required to register an `EventListener` with a GUI object. This chapter also explains how to have events generated by GUI objects located in one class handled by an `EventListener` located in another class. This chapter also presents a detailed discussion of how to handle GUI events using inner and anonymous classes.

CHAPTER 14 - AN ADVANCED GUI PROJECT

Chapter 14 builds on chapters 12 and 13 by walking readers through the creation of a significant Java GUI project. It shows readers how Swing paints components, how to use the `Graphics` class, how to load resources from a

package-relative path, how to write custom renderers, and how to write a custom editor. This chapter also details how to create custom Events and EventListeners, how to create an offscreen image, how to paint with transparency, and how to extend Swing components to provide significant new functionality.

PART IV: INTERMEDIATE CONCEPTS

Part IV contains four chapters that deal with exceptions, threads, collections, and file input and output. This material is considered intermediate because it is assumed at this point that readers are familiar with the core Java language features presented in part II. The material in this section should be well-understood by readers before they attempt part V - Network Programming.

CHAPTER 15 - EXCEPTIONS

Chapter 15 dives deeply into the subject of exceptions. It begins by discussing the Throwable class hierarchy and explains the difference between errors, exceptions, runtime exceptions, and checked vs. unchecked exceptions. It then moves into a detailed discussion of exception handling using different combinations of the try-catch-finally blocks. The chapter then shows readers how to throw exceptions, create custom exceptions, and how to translate low-level exceptions into higher-level exception abstractions more appropriate for their program needs.

CHAPTER 16 - THREADS

Chapter 16 focuses on the subject of thread programming. It starts with the definition of a thread and describes the different ways to start and stop a thread. It then describes the difference between extending the Thread class and implementing the Runnable interface. This chapter also explains how to set thread priorities. It correlates the behavior of thread priority to popular operating systems and explains how thread race conditions can occur. It also clearly explains the mechanics of thread synchronization and shows readers how to manage multiple threads that share common resources. The chapter concludes by showing readers how to use the Java thread wait and notify mechanism to ensure cooperation between threads.

CHAPTER 17 - COLLECTIONS

Chapter 17 gives readers a solid overview of the Java collection framework. The chapter begins by explaining the rationale for a collections framework by developing a home-grown dynamic array class. It then presents an overview of the Java collections framework followed by a discussion of Java 1.4 collections. The chapter concludes with a discussion of Java 5 generics and shows readers how to use generic collection classes as well as how to write their own generic methods.

CHAPTER 18 - FILE I/O

Chapter 18 offers a fairly complete treatment of file I/O programming using the classes from the java.io package. The chapter shows readers how to make sense out of what is considered to be the most confusing package in the Java platform API. It then goes on to demonstrate the use of every class in the java.io package that can be used for file I/O. Although the focus of the chapter is on file I/O, the information learned here can be directly applied when programming network I/O operations. The chapter concludes with a comprehensive programming example showing how the RandomAccessFile class can be used to create a legacy datafile adapter.

PART V: NETWORK PROGRAMMING

Part V consists of three chapters that cover a broad range of network programming concepts to include an introduction to networking and distributed applications, client-server applications, Remote Method Invocation (RMI) programming, Applets and JDBC.

CHAPTER 19 – INTRODUCTION TO NETWORKING AND DISTRIBUTED APPLICATIONS

Chapter 19 provides a broad introduction to the concepts associated with network programming and distributed applications. It begins by exploring the meaning of the term computer network and shows how the TCP/IP network protocols serve as the foundation protocols for internet programming. The chapter continues with a discussion of client-server computing, application distribution, multi-tiered applications, and physical vs. logical application tier distribution. It explains clearly how multiple JVMs must be deployed in order to execute physically-distributed Java applications. The chapter then goes on to explain the purpose and importance of the internet protocols TCP/IP. The chapter concludes with an RMI programming example on both the Java 1.4 and Java 5 platforms.

CHAPTER 20 – CLIENT-SERVER APPLICATIONS

Chapter 20 focuses on client-server application programming and steps the reader through a comprehensive network programming example that utilizes both RMI and I/O streams. It explains and demonstrates the use of the `ServerSocket`, `Socket`, and `Thread` classes to write multi-threaded client-server applications. The chapter also explains how to use the `DataInputStream` and `DataOutputStream` classes to send data between socket-based client-server applications. Additional topics discussed in the chapter include the use of the `Properties` class to store and retrieve application properties, how to formulate proprietary network application protocols, how to use the `Class.forName()` method to dynamically load classes at application runtime, and how to use the singleton and factory design patterns in a program.

CHAPTER 21 – APPLETS AND JDBC

Chapter 21 provides a discussion of applets and Java Database Connectivity (JDBC) programming. It begins with a treatment of applets, their purpose and use, and their lifecycle stages. It discusses applet security restrictions and shows readers how to design with these security restrictions in mind. The applet discussion continues by showing readers how to pass information into an applet from an HTML page via applet parameters. The chapter then presents a discussion of JDBC and shows readers how to store data to and retrieve data from a relational database via the JDBC API. The chapter concludes with a comprehensive programming example showing readers how to write an applet that serves as a front-end to an RMI-based application that uses JDBC to manipulate data stored in a MySQL database.

PART VI: OBJECT-ORIENTED PROGRAMMING

Part VI contains four chapters and presents material related to the theoretical aspects of object-oriented design and programming. Topics include a deeper treatment of inheritance, composition, interfaces, polymorphic behavior, how to create well-behaved Java objects, three design principles, and a formal introduction to a few helpful design patterns.

CHAPTER 22 – INHERITANCE, COMPOSITION, INTERFACES, AND POLYMORPHISM

Chapter 22 dives deeper into the topics of inheritance, compositional design, the role of interfaces, and how to achieve polymorphic behavior. It begins by comparing the goals of inheritance vs. composition and suggests that the true goal of good design is to know when the design is good enough by recognizing and making rational software engineering tradeoffs. The chapter continues by giving expanded coverage of inheritance concepts and introduces readers to Meyer's inheritance taxonomy and Coad's inheritance criteria. Next, the chapter explores the role of interfaces and how they can be used to reduce or eliminate intermodule dependencies. It introduces the concept of modeling dominant, collateral, and dynamic roles and suggests when interfaces might be used vs. a class hierarchy. The chapter concludes with a discussion of applied polymorphic behavior and composition-based design as a force multiplier.

CHAPTER 23 – Well-Behaved Objects

Chapter 23 shows readers how to create well-behaved Java objects by correctly overriding the methods provided by the `java.lang.Object` class. These methods include the `toString()`, `equals()`, `hashCode()`, and `clone()` methods. The chapter shows readers how to implement two different hash code algorithms and discusses the difference between a deep vs. a shallow copy. The chapter continues by showing readers how to implement both the `java.lang.Comparable` and the `java.util.Comparator` interfaces. It then concludes by demonstrating the use of well-behaved objects in a collection.

CHAPTER 24 – Three Design Principles

Chapter 24 introduces readers to three important object-oriented design principles that can be immediately applied to improve the quality of their application architectures. The chapter begins with a discussion of the preferred characteristics of an object-oriented architecture. It then presents a discussion of the Liskov substitution principle and Bertrand Meyer’s design-by-contract, preconditions, postconditions, and class invariants. This is followed with a discussion of the open-closed principle and the dependency inversion principle.

CHAPTER 25 – Helpful Design Patterns

Chapter 25 presents a formal introduction to the topic of software design patterns. It begins by describing the purpose and use of design patterns and their origin. The chapter then discusses and demonstrates the singleton, factory, model-view-controller, and command patterns. The comprehensive programming example presented at the end of the chapter demonstrates the use of all these patterns in one application.

Pedagogy

Each chapter takes the following structure:

- Learning Objectives
- Introduction
- Content
- Quick Reviews
- Summary
- Skill-Building Exercises
- Suggested Projects
- Self-Test Questions
- References
- Notes

LEARNING OBJECTIVES

Each chapter begins with a set of learning objectives. The learning objectives specify the minimum knowledge gained by reading the chapter material and completing the skill-building exercises, suggested projects, and self-test questions. In almost all cases the material presented in each chapter exceeds the chapter learning objectives.

INTRODUCTION

The introduction provides a context and motivation for reading the chapter material.

CONTENT

The chapter content represents the core material. Core material is presented in sections and sub-sections.

Quick Reviews

The main points of each primary section heading are summarized in a quick review section. A quick review may be omitted from relatively short sections.

SUMMARY

The summary section summarizes the chapter material.

Skill-Building EXERCISES

Skill-building exercises are small programming or other activities intended to strengthen your Java programming capabilities in a particular area. They could be considered focused micro-projects.

SUGGESTED PROJECTS

Suggested projects require the application of a combination of all knowledge and skills learned up to and including the current chapter to complete. Suggested projects offer varying degrees of difficulty.

SELF-TEST QUESTIONS

Self-test questions test your comprehension on material presented in the current chapter. Self-test questions are directly related to the chapter learning objectives. Answers to all self-test questions appear in appendix C which is included on the supplemental CD-ROM and the Pulp Free Press website.

REFERENCES

All references used in preparing chapter material are listed in the references section.

NOTES

Note taking space.

TYPOGRAPHICAL FORMATS

The preceding text is an example of a primary section heading. It is set in Peynot font at 14 point normal with lines above and below.

THIS IS AN EXAMPLE OF A FIRST-LEVEL SUBHEADING

It is set in Peynot font at 12 point normal.

THIS IS AN EXAMPLE OF A SECOND-LEVEL SUBHEADING

It is set in Peynot font at 10 point oblique.

THIS IS AN EXAMPLE OF A THIRD-LEVEL SUBHEADING

It is set in Peynot font at 9 point, oblique, underline, and indented to the right.

THIS IS AN EXAMPLE OF A FOURTH-LEVEL SUBHEADING

It is set in Peynot font at 9 point, regular, and indented a little more to the right.

SOURCE CODE FORMATTING

Source code and other example listings appear as line-numbered paragraphs set in Courier New font. Each line of code has a distinct number. Long lines of code are allowed to wrap as is shown in the following example:

```
1      // this is a line of code
2      // this is a line of code
3      // this is a really long line of code that cannot be split along a natural boundary. Lines such as these
are allowed to wrap to the next line. Long lines of code such as these have been kept to a minimum.
4      // this is another line of code
```

CD-ROM

The *Java For Artists* supplemental CD-ROM includes the following items:

- The complete source code for all programming examples offered in the text
- The PDF edition of *Java For Artists*
- Sun's Java™ 2 Software Development Kit, Standard Edition, Version 1.4.2 for Microsoft Windows and Linux
- Sun's Java™ 2 Platform Standard Edition Development Kit 5.0 for Microsoft Windows and Linux
- Evaluation version of Helios Software Solution's TextPad programmer's text editor for Microsoft Windows
- Adobe® Acrobat® Reader Version 7 for Windows 2000 Service Packs 2 through 4 and Linux
- Answers to all Self-Test questions

SUPPORTSITE™ WEBSITE

The *Java For Artists* SupportSite™ is located at [<http://pulpfreepress.com/SupportSites/JavaForArtists/>]. The support site includes source code, links to Java development and UML tool vendors and corrections and updates to the text.

PROBLEM REPORTING

Although every possible effort was made to produce a work of superior quality some mistakes will no doubt go undetected. All typos, misspellings, inconsistencies, or other problems found in *Java For Artists* are mine and mine alone. To report a problem or issue with the text please contact me directly at rick@pulpfreepress.com or report the problem via the *Java For Artists* SupportSite™. I will happily acknowledge your assistance in the improvement of this book both online and in subsequent editions.

ABOUT THE AUTHORS

Rick Miller is a senior computer scientist and web applications architect for Science Applications International Corporation (SAIC) where he designs and builds enterprise web applications for the Department of Defense intelligence community. He holds a master's degree in computer science from California State University, Long Beach and is an assistant professor at Northern Virginia Community College, Annandale Campus, where he teaches a variety of computer programming courses. He enjoys reading, writing, photography, and rummaging through used book shops. A small sampling of his photos can be found at www.warrenworks.com. Rick is a certified Java programmer and developer.

Raffi Kasparian holds his degrees in Music Performance, the highest of which is Doctor of Musical Arts (DMA) from The University of Michigan. He is the Pianist for the Men's Chorus of the United States Army Band and is also a part time software engineer for Science Applications International Corporation (SAIC). Until he decided at the age of 14 to seriously pursue classical piano, he had intended to become a mathematician or scientist. After settling into his position with the Army, he revived his love of science and picked up computer programming as a passionate hobby. He has developed applications ranging from 3D scientific visualization systems to enterprise level database and web applications. Raffi is a certified Java programmer and developer.

Acknowledgments

Raffi and I would like to personally thank the following folks for devoting their precious time reviewing and commenting on various portions of the text: Derek Ashmore, Jim Brox, John Cosby, Adrienne Fargas, Dave Harper, Stephenie Husband, Jeff Meyer, Alex Remily, and Brendan Richards. The help of such friends makes anything possible.

A handwritten signature in black ink, consisting of a large, stylized 'R' followed by a 'M' and a long horizontal flourish extending to the right.

Rick Miller
Falls Church, Virginia

PART I: THE JAVA STUDENT SURVIVAL GUIDE

CHAPTER 1



THREE ON A BEACH

AN APPROACH TO THE ART OF PROGRAMMING

LEARNING OBJECTIVES

- *Describe the difficulties you will encounter in your quest to become a JAVA™ PROGRAMMER*
- *List and describe the features of an INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)*
- *List and describe the stages of the “Flow”*
- *List and describe the three roles you will play as a programming student*
- *State the purpose of the project-approach strategy*
- *List and describe the steps of the project-approach strategy*
- *List and describe the steps of the development cycle*
- *List and describe two types of project complexity*
- *State the meaning of the terms “maximize cohesion” and “minimize coupling”*
- *Describe the differences between functional decomposition and object-oriented design*
- *State the meaning of the term “isomorphic mapping”*
- *State the importance of writing self-commenting code*
- *State the purpose and use of the three types of Java comments*

INTRODUCTION

Programming is an art; there's no doubt about it. Good programmers are artists in every sense of the word. They are a creative bunch, although some would believe themselves otherwise out of modesty. As with any art, you can learn the secrets of the craft. That is what this chapter is all about.

Perhaps the most prevalent personality trait I have noticed in good programmers is a knack for problem solving. Problem solving requires creativity, and lots of it. When you program a computer you are solving a problem with a machine. You transfer your knowledge of a particular problem into code, transform the code into a form understandable by a machine, and run the result on a machine. Doing this requires lots of creativity, especially when you find yourself stumped by a particular problem.

The material presented here is wrought from experience. Believe it or not, the hardest part about learning to program a computer, in any programming language, is not the learning of the language itself; rather, it is learning how to approach the art of problem solving with a computer. To this end, the material in this chapter is aimed squarely at the beginner. However, I must issue a word of warning. If you are truly a novice, then some of what you read in this chapter will make less sense to you than to someone already familiar with Java. Don't worry, it's that way by design. If you feel like skipping parts of this chapter now, then go right ahead. The material will be here when you need it. In fact, you will grow to appreciate this chapter more as you gain experience as a programmer.

THE DIFFICULTIES YOU WILL ENCOUNTER LEARNING JAVA

During your studies of the Java programming language you will face many challenges and frustrations. However, the biggest problem you will encounter is not the learning of the language itself, but the many other skills and tools you must learn before writing programs of any significance or gaining any measure of proficiency in solving problems with Java. If you are a seasoned student or practicing computer professional returning to the classroom to upgrade your skills, you have the advantage of experience. You can concentrate on learning the syntax and nuances of Java and very quickly apply its powers to problems at hand. If you are an absolute beginner, however, you have much to learn.

Required Skills

In addition to the syntax and semantics of the Java language you will need to master the following skills and tools:

- A development environment, which could be as simple as a combination of a text editor and compiler or as complex as a commercial product that integrates editing, compiling, and project management capabilities into one suite of tools
- A computing platform of choice (*i.e.*, an Apple Macintosh or Microsoft Windows machine)
- Problem solving skills
- How to approach a programming project
- How to manage project complexity
- How to put yourself in the mood to program
- How to stimulate your creative abilities
- Object-oriented analysis and design
- Object-oriented programming principles
- Java platform Application Programming Interface (API)

The Planets Will Come Into Alignment

I use a metaphor to describe what it takes before you can get even the simplest program to execute properly. It's as if the planets must come into alignment. You must learn a little of each skill and tool listed above, with the exception of object-oriented programming principles and object-oriented analysis and design, to write, compile, and run your first Java program. But, when the planets do come into alignment, and you see your first program compile and

execute, and you begin to make sense of all the class notes, documentation, and text books you have studied up to that point, you will spring up from your chair and do a victory dance. It's a great feeling!

How This CHAPTER Will Help You

This chapter gives you the information you need to bring the planets into alignment sooner rather than later. It presents an abbreviated software development methodology that formalizes the three primary roles you play as a programming student: analyst, architect, and programmer. It offers tips on how you can tap into the “flow” which is a transcendental state often experienced by artists when they are completely absorbed in, and focused on, their work. It also offers several strategies to help you manage project complexity, something you will not need to do for very small projects but should get into the habit of doing as soon as possible.

I recommend you read this chapter at least once in its entirety, and refer back to it as necessary as you progress through the text.

PROJECT MANAGEMENT

THREE SOFTWARE DEVELOPMENT ROLES

You will find yourself assuming the duties and responsibilities of three software development roles: analyst, architect, and programmer.

Analyst

The first software development role you will play as a student is that of analyst. When you are first handed a class programming project you may not understand what, exactly, the instructor is asking you to do. Hey, it happens! Regardless, you, as the student, must read the assignment and design and implement a solution.

Programming project assignments come in several flavors. Some instructors go into painful detail about how they want the student to execute the project. Others prefer to generally describe the type of program they want, thus leaving the details, and the creativity, up to you. There is no one correct method of writing a project assignment; each has its benefits and limitations.

A detailed assignment takes a lot of the guesswork out of what outcome the instructor expects. On the other hand, having every design decision made for you may prevent you from solving the problem in a unique, creative way.

A general project assignment delegates a lot of decision making to the student while also adding the responsibility of determining what project features will satisfy the assignment.

Both types of assignments model the real world to some extent. Sometimes software requirements are well defined and there is little doubt what shape the final product will take and how it must perform. More often than not, however, requirements are ill-defined and vaguely worded. As an analyst you must clarify what is being asked of you. In an academic setting, do this by talking to the instructor and ask him to clarify the assignment. A clear understanding of the assignment will yield valuable insight into possible approaches to a solution.

Architect

The second software development role you will play is that of architect. Once you understand the assignment you must design a solution. If your project is extremely small you could perhaps skip this step with no problem. However, if your project contains several objects that interact with each other, then your design, and the foundation it lays, could make the difference between success and failure. A well-designed project reflects a sublime quality that poorly designed projects do not.

Two objectives of good design are the abilities to accommodate change and tame complexity. Change, in this context, means the ability to incrementally add features to your project as it grows without breaking the code you have already written. Several important object-oriented principles have been formulated to help tame complexity and

will be discussed later in the book. For starters though, begin by imposing a good organization upon your source-code files. For simple projects you can group related project source-code files together in one directory. For more complex projects you will want to organize your source-code files into packages. I will discuss packages later in the chapter.

PROGRAMMER

The third software development role you will play is that of programmer. As the programmer you will execute your design. The important thing to note here is that if you do a poor job in the roles of analyst and architect, your life as a programmer will be miserable. That doesn't mean the design has to be perfect. I will show you how to incrementally develop and make improvements to your design as you code.

Now that you know what roles you will play as a student, let's discuss how you might approach a project.

A PROJECT-APPROACH STRATEGY

Most students have difficulty implementing their first significant programming assignment, not because they lack brains or talent, but because they lack experience. If you are a novice and feel overwhelmed by your first programming project, rest assured you are not alone. The good news is that with practice, and some small victories, you will quickly gain proficiency at formulating approach strategies to your programming projects.

Even experienced programmers may not immediately know how to solve a problem or write a particular piece of code when tasked to do so. What they do know, however, is how to formulate a strategy to solve the problem.

YOU HAVE BEEN HANDED A PROJECT – NOW WHAT?

Until you gain experience and confidence in your programming abilities, the biggest problem you will face when given a large programming assignment is where to begin. What you need to help you in this situation is a project-approach strategy. The strategy is presented below and discussed in detail. I have also summarized the strategy in a checklist located in appendix A. Feel free to reproduce the checklist to use as required.

The project-approach strategy is a collection of areas of concern to take into consideration when you begin a programming project. It's not a hard, fast list of steps you must take. It's intended to put you in control, to point you in the right direction, and give you food for thought. It is flexible. You will not have to consider every area of concern for every project. After you have used it a few times to get you started you may never use it explicitly again. As your programming experience grows, feel free to tailor the project-approach strategy to suit your needs.

STRATEGY AREAS of CONCERN

The project-approach strategy consists of several programming project *areas of concern*. These areas of concern include application requirements, problem domain, language features, and application design. When you use the strategy to help you solve a programming problem your efforts become focused and organized rather than ad hoc and confused. You will feel like you are making real progress rather than drowning in a sea of confusion.

APPLICATION REQUIREMENTS

An application requirement is an assertion about a particular aspect of expected application behavior. A project's application requirements are contained in a project specification or programming assignment. Before you proceed with the project you must ensure that you completely understand the project specification. Seek clarification if you do not know, or if you are not sure, what problem the project specification is asking you to solve. In my academic career I have seen projects so badly written that I thought I had a comprehension problem. I'd read the thing over and over again until struck by a sudden flash of inspiration. But more often than not, I would reinforce what I believed an instructor required by asking them to clarify any points I did not understand.

PROBLEM DOMAIN

The problem domain is the specific problem you are tasked to solve. I would say that it is that body of knowledge necessary to implement a software solution apart and distinct from the knowledge of programming itself. For

instance, “Write a program to simulate elevator usage in a skyscraper.” You may understand what is being asked of you (*requirements understanding*) but not know anything about elevators, skyscrapers, or simulations (*problem domain*). You need to become enough of an expert in the problem domain you are solving so that you understand the issues involved. In the real world, subject matter experts (SMEs) augment development teams, when necessary, to help developers understand complex problem domains.

PROGRAMMING LANGUAGE FEATURES

One source of great frustration to novice students at this stage of the project is knowing what to design but not knowing enough of the language features to start the design process. This is when panic sets in and students begin to buy extra books in hopes of discovering the Holy Grail of project wisdom.

To save yourself from panic, make a list of the language features you need to understand and study each one, marking it off your list as you go. This provides focus and a sense of progress. As you read about each feature take notes on its usage and refer to your notes when you sit down to formulate your program’s design.

HIGH-LEVEL DESIGN & IMPLEMENTATION STRATEGY

When you are ready to design a solution you will usually be forced to think along two completely different lines of thought: procedural vs. object-oriented.

PROCEDURAL-BASED DESIGN APPROACH

A procedural-based design approach is one in which you identify and implement program data structures separately from the program code that manipulates those data structures. When taking a procedural approach to a solution you generally break the problem into small, easily solvable pieces, implement the solution to each of the pieces separately, and then combine the solved pieces into a complete solution. The solvable pieces I refer to here are called functions. This methodology is also known as functional decomposition.

Although Java does not support stand-alone functions (*Java has methods and a method must belong to a class*), a procedural-based design approach can be used to create a working Java program, although taking such an approach usually results in a sub-optimal design.

OBJECT-ORIENTED DESIGN APPROACH

Object-oriented design entails thinking of an application in terms of objects and the interactions between these objects. No longer are data structures and the methods that manipulate those data structures considered to be separate. The data an object needs to do its work is contained within the object itself and resides behind a set of public interface methods. Data structures and the methods that manipulate them combine to form classes from which objects can then be created.

A problem solved with an object-oriented approach is decomposed into a set of objects and their associated behavior. Design tools such as the Unified Modeling Language (UML) can be used to help with this task. Following the identification of system objects, object interface methods must then be defined. Classes must then be declared and defined to implement the interface methods. Following the implementation of all application classes, they are combined and used together to form the final program. (*This usually takes place in an iterative fashion over a period of time according to a well-defined development process.*) Note that when using the object-oriented approach you are still breaking a problem into solvable pieces, only now the solvable pieces are objects that represent the interrelated parts of a system.

The primary reason the object-oriented approach is superior to functional decomposition is due to the isomorphic mapping between the problem domain and the design domain as figure 1-1 illustrates. Referring to figure 1-1, real-world objects such as weapon systems, radars, propulsion systems, and vessels have a corresponding representation in the software system design. The correlation between real-world objects and software design components, and ultimately to the actual code modules, fuels the power of the object-oriented approach.

Once you get the hang of object-oriented design you will never return to functional decomposition again. However, after having identified the objects in your program and the interfaces they should have, you must implement your design. This means writing class member methods one line of code at a time.

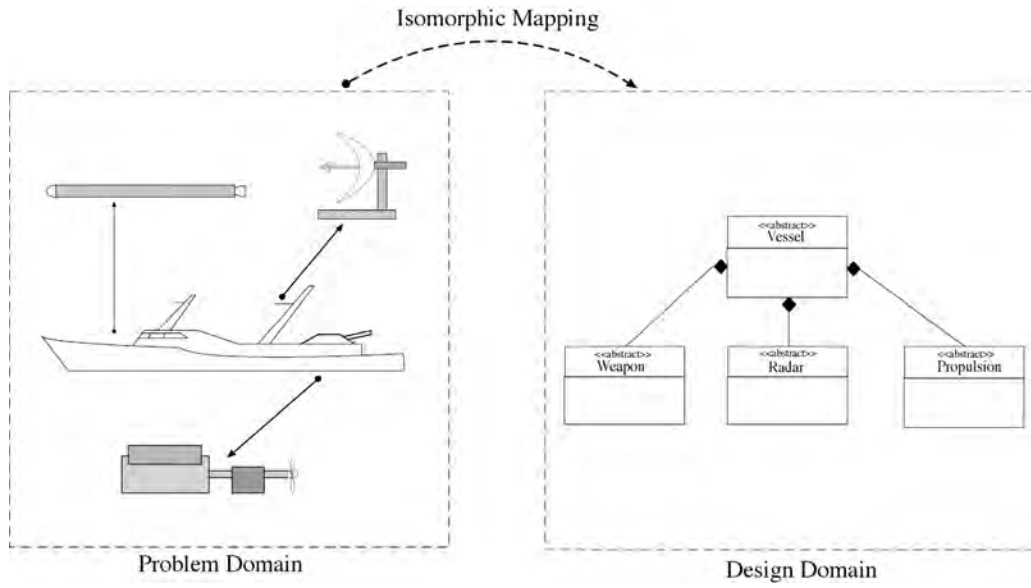


Figure 1-1: Isomorphic Mapping Between Problem Domain and Design Domain

Think Abstractly

One mistake students often make is to think too literally. It is very important to remember that the act of solving a real-world problem with a computer requires abstraction. The real world is too complex to model sufficiently with a computer program. One day, perhaps, the human race will produce a genius who will show us how it's done. Until then, analysts must focus on the essence of a problem and distill unnecessary details into a tractable solution that can then be modeled effectively in software.

THE STRATEGY IN A NUTSHELL

Identify the problem, understand the problem, make a list of language features you need to study and check them off as you go. Once you formulate a solution to the problem, break the problem into manageable pieces, solve each piece of the problem, and then combine the solved pieces to form a total solution.

Applicability To THE REAL WORLD

The problem-approach strategy presented above is not intended to replace a formal course on software engineering, but it will help you when you enter the real world as a commercial programmer. In that world, you will soon discover that all companies and projects are not created equal. Different companies have different software development methodologies. Some companies have no software development methodology. If you find yourself working for such a company you will probably be the software engineering expert. Good luck!

THE ART OF PROGRAMMING

Programming is an art. Any programmer will agree — it takes a lot of creativity to solve problems with a computer. Creative people have an advantage in that they are not afraid to explore new avenues of design. Their open-mindedness and readiness to accept new ideas gives them the ability to see problems differently from people who tend towards the cut and dry. This section offers a few suggestions on how you can stimulate your creativity.

DON'T START AT THE COMPUTER

Unless you have a good idea about what source code to write, sitting down at the computer without first thinking through some design issues is the worst mistake you can make. If you have ever suffered from writer's block when writing a paper for class then you can begin to understand what you will experience if you begin your project at the computer.

I recommend you forget the computer, go someplace quiet and relaxing, with pen and paper, and draft a design document. It doesn't have to be big or too detailed. Entire system designs can be sketched on the back of a napkin. The important thing is that you give some prior thought regarding your program's design and structure before you start coding.

Your choice of relaxing locations is important. It should be someplace where you feel really comfortable. If you like quiet spaces, then seek quiet spaces; if you like to watch people walk by and think of the world, then an outdoor cafe may be the place for you. Inside, outside, at the beach, on the ski slope, wherever you prefer.

What you seek is the ability to let your mind grind away on the solution. Let your mind do the work. Writing code at the computer is a mechanical process. Formulating the solution is where real creativity is required and is the part of the process that requires the most brainpower. Typing code is more like an exercise on attention to detail.

INSPIRATION STRIKES AT THE WEIRDEST TIME

If you let your mind work on the problem, it will offer its solution to you at the weirdest times. I solve most of my programming problems in my sleep. As a student, I kept computers in the bedroom and would get up at all hours of the night to work on ideas that had popped into my head in a dream.

Try to have something to write on close at hand at all times. A pad of paper and pen next to the bed or next to the toilet can come in handy! You can also use a small tape recorder, digital memo recorder, or your personal digital assistant. Whatever means suit your style. Just be prepared. There's nothing worse than the sinking feeling of having had the solution come to you in the middle of the night, or in the shower, or on the drive home from work or school, only to forget it later. You'll be surprised at how many times you'll say to yourself, "*Hey, that will work!*" only to forget it and have no clue what you were thinking when you finally get hold of a pen.

OWN YOUR OWN COMPUTER

Do not rely on the computer lab! I repeat, do not rely on the computer lab! The computer lab is the worst possible place for inspiration and cranking out code. If at all possible, you should own your own computer. It should be one that is sufficiently powerful enough to be used for Java software development.

YOU EITHER HAVE TIME AND NO MONEY, OR MONEY AND NO TIME

The one good reason for not having your own personal computer is severe economic hardship. Full-time students sometimes fall into this category. If you are a full-time student then what you usually have instead of a job, or money, is gobs of time. So much time that you can afford to spend your entire day at school and complain to your friends about not having a social life. But you can stay in the computer lab all day long and even be there when it is relatively quiet.

On the other hand, you may work full-time and be a part-time student. If this describes you, then you don't have time to screw around driving to school to use the computer lab. You will gladly pay for any book or software package that makes your life easier and saves you time.

THE FAMILY COMPUTER IS NOT GOING TO CUT IT!

If you are a family person working full-time and attending school part-time, then your time is a precious commodity. If you have a family computer that everyone shares, adults as well as children, then get another computer, put it off limits to everyone but yourself, and password-protect it. This will ensure that your loving family does not accidentally wipe out your project the night before it is due. Don't kid yourself, it happens. Ensure your peace of mind by having your own computer in your own little space with a sign on it that reads, "*Touch This Computer And Die!*"

SET THE MOOD

When you have a good idea on how to proceed with entering source code, you will want to set the proper programming mood.

LOCATION, LOCATION, LOCATION

Locate your computer work area someplace that's free from distraction. If you are single, this may be easier than if you are married with children. If you live in a dorm or frat house, good luck! Perhaps the computer lab is an alternative after all.

Have your own room if possible, or at least your own corner of a larger room that is recognized as a quiet zone. Noise-canceling headphones might help if you find yourself in this situation.

Set rules. Let your friends and family know that it's not cool to bother you when you are programming. I know it sounds rude, but when you get into the *flow*, which is discussed below, you will become agitated when someone interrupts your train of thought to ask you about school lunch tomorrow or the location of the car keys. Establish the ground rules up front that say when it is a good time to disturb you when you are programming. The rule is - never!

CONCEPT OF THE FLOW

Artists tend to become absorbed in their work, not eating and ignoring personal hygiene for days, even weeks, at a time. Those who have experienced such periods of intense concentration and work describe it as a transcendental state where they have complete clarity of the idea of the finished product. They tune out the world around them, living inside a cocoon of thought and energy.

Programmers can get into the flow. I have achieved the flow. You too can achieve the flow. When you do, you will crave the feeling of the flow again. It is a good feeling, one of complete and utter understanding of what you are doing and where you are going with your source code. You can do amazing amounts of programming while in the flow.

THE STAGES OF FLOW

Like sleep, there are stages to the flow.

GETTING SITUATED

The first stage. You sit down at the computer and adjust your keyboard and stuff around you. Take a few deep breaths to help you relax. By now you should have a good idea of how to proceed with your coding. If not, you shouldn't be sitting at the computer.

RESTLESSNESS

Second stage. You may find it difficult to clear your mind of the everyday thoughts that block your creativity and energy. Maybe you had a bad day at work, or a great day. Perhaps your spouse or significant other is being a complete jerk! Perhaps he or she is being especially nice and you are wondering why.

Close your eyes, breathe deeply and regularly. Clear your mind and think of nothing. It is hard to do at first, but with practice it becomes easy. When you can clear your mind and free yourself from distracting thoughts, you will find yourself ready to begin coding.

SETTLING IN

Now your mind is clear. Non-productive thoughts are tucked neatly away. You begin to program. Line by line your program takes shape. You settle in, and the clarity of your purpose takes hold and propels you forward.

CALM AND COMPLETE FOCUS

You don't notice it at first, but at some point between this and the previous stage you have slipped into a deeply relaxed state, and are utterly focused on the task at hand. It is like reading a book and becoming completely absorbed. Someone can call your name, but you will not notice. You will not respond until they either shout at you or do something to break your concentration.

You know you were in the flow, if only to a small degree, when being interrupted brings you out of this focused state, and you feel agitated and have to settle in once again. If you avoid doing things like getting up from your chair for fear of breaking your concentration or losing your thought process, then you are in the flow!

BE EXTREME

Kent Beck, in his book "*Extreme Programming Explained*", describes the joy of doing really good programming. The following programming cycle is synthesized from his extreme programming philosophy.

THE PROGRAMMING CYCLE

PLAN

Plan a little. Your project design should serve as a guide in your programming efforts. Your design should also be flexible and accommodate change. This means that as you program, you may make changes to the design.

Essentially, you will want to design to the point where you have enough of the design to allow you to begin coding. The act of coding will either soon reinforce your design decisions, or detect fatal flaws that you must correct if you hope to have a polished, finished project.

CODE

Code a little. Write code in small, cohesive modules. A class or method at a time usually works well.

TEST

Test a lot. Test each class, module, or method either separately or in whatever grouping makes sense. You will find yourself writing little programs on the side called *test cases* to test the code you have written. This is a good practice to get into. A test case is nothing more than a little program you write and execute in order to test the functionality of some component or feature before integrating that component or feature into your project. The objective of testing is to break your code and correct its flaws before it has a chance to break your project in ways that are hard to detect.

INTEGRATE/TEST

Integrate often, and perform regression testing. Once you have a tested module of code, be it either a method or complete set of related classes, integrate the tested component(s) into your project regularly. The objective of regular integration and regression testing is to see if the newly integrated component or newly developed functionality breaks any previously tested and integrated component(s) or integrated functionality. If it does, then remove it from the project and fix the problem. If a newly integrated component breaks something you may have discovered a design flaw or a previously undocumented dependency between components. If this is the case then the next step in the programming cycle should be performed.

REFACTOR

Refactor the design when possible. If you discover design flaws or ways to improve the design of your project, you should refactor the design to accommodate further development. An example of design refactoring might be the migration of common elements from derived classes into the base class to take better advantage of code reuse.

REPEAT

Apply the programming cycle in a tight spiral fashion. You will quickly reach a point in your project where it all starts to come together, and very quickly so.

The Programming Cycle Summarized

Plan a little, code a little, test a lot, integrate often, refactor the design when possible. **Don't Wait Until You Think You Are Finished Coding The Entire Project To Compile!** Trying to write the entire program before compiling a single line of code is the most frequent mistake new programmers tend to make. The best advice I can offer is don't do it! Use the programming cycle previously outlined. Nothing will depress you more than seeing a million compiler errors scroll up the screen.

A Helpful Trick: Stubbing

Stubbing is a programmer's trick you can use to speed development and avoid having to write a ton of code just to get something useful to compile. Stubbing is best illustrated by example.

Say that your project requires you to display a text-based menu of program features on the screen. The user would then choose one of the menu items and press ENTER, thereby invoking that menu command. What you would really like to do is write and test the menu's display and selection methods without worrying about having it actually perform the indicated action. You can do exactly that with stubbing.

A stubbed method, in its simplest form, is a method with an empty body. It's also common to have a stubbed method display a simple message to the screen saying in effect, *"Yep, the program works great up to this point. If it were actually implemented you'd be using this feature right now!"*

Stubbing is a great way to incrementally develop your project. Stubbing will change your life!

Fix The First Compiler Error First

OK. You compile some source code and it results in a slew of compiler errors. What should you do? I recommend you stay calm, take a deep breath, and fix the first compiler error first. Not the easiest compiler error, but the first compiler error. The reason is that the first error detected by the compiler, if fatal, will generate other compiler errors. Fix the first one first, and you will generally find a lot of the other errors will also be resolved. If you pick an error from the middle of the pack and fix it, you may introduce more errors into your source code. Fix the first compiler error first!

MANAGING PROJECT COMPLEXITY

Software engineers generally encounter two types of project complexity: conceptual and physical. All programming projects exhibit both types of complexity to a certain degree, but the approach and technique used to manage small-project complexity will prove woefully inadequate when applied to medium, large, or extremely large programming projects. This section discusses both types of complexity, and suggests an approach for the management of each.

CONCEPTUAL COMPLEXITY

Conceptual complexity is that aspect of a software system that is manifested in, dictated by, and controlled by its architectural design. A software architectural design is a specification of how each software module or component will interact with the other software components. A project's architectural design is the direct result of a solution approach conceived of by one or more software engineers in an attempt to implement a software solution to a particular problem domain. In formulating this solution, the software engineers are influenced by their education and experience, available technology, and project constraints.

An engineer versed in procedural programming and functional decomposition techniques will approach the solution to a programming problem differently from an engineer versed in object-oriented analysis and design techniques. For example, if a database programmer versed in stored-procedure programming challenged an Enterprise Java programmer to develop a Web application, the two would go about the task entirely differently. The database programmer would place business logic in database stored procedures as well as in the code that created the application web pages. The Enterprise Java programmer, on the other hand, would create a multi-tiered application. He would isolate web-page code in the web-tier, and isolate business logic in the business-tier. He would create a separate data-access tier to support the business-tier and use the services of a database application to persist business objects.

By complete chance, both applications might turn out looking exactly the same from a user's perspective. But if the time ever came to scale the application up to handle more users, or to add different application functionality, then the object-oriented, multi-tiered approach would prove superior to the database-centric, procedural approach. Why?

The answer lies in whether a software architecture is structured in such a way that makes it receptive and resilient to change. Generally speaking, procedural-based software architectures are not easy to change whereas object-oriented software architectures are usually more so. The co-mingling of business logic with presentation logic, and the tight coupling between business logic and its supporting database structures, renders the architecture produced by the database programmer difficult to understand and change. The object-oriented approach lends itself, more naturally, to the creation of software architectures that support the separation of concerns. In this example, the web-tier has different concerns than the business-tier, which has separate concerns from the data-access tier, etc.

However, writing a program in Java, or in any other object-oriented programming language does not, by default, result in a good object-oriented architecture. It takes lots of training and practice to develop good, robust, change-receptive and resilient software architectures.

MANAGING CONCEPTUAL COMPLEXITY

Conceptual complexity can either be tamed by a good software architecture, or it can be aggravated by a poor one. Software architectures that seem to work well for small to medium-sized projects will be difficult to implement and maintain when applied to large or extremely large projects.

Conceptual complexity is tamed by applying sound object-oriented analysis and design principles and techniques to formulate robust software architectures that are well-suited to accommodate change. Well-formulated object-oriented software architectures are much easier to maintain compared to procedural-based architectures of similar or smaller size. That's right — large, well-designed object-oriented software architectures are easier to maintain and extend than small, well-designed procedural-based architectures. The primary reason for this fact is that it's easier for object-oriented programmers to get their heads around an object-oriented design than it is for programmers of any school of thought to get their heads around a procedural-based design.

THE UNIFIED MODELING LANGUAGE (UML)

The Unified Modeling Language (UML) is the de facto standard modeling language of object-oriented software engineers. The UML provides several types of diagrams to employ during various phases of the software development process such as use-case, component, class, and sequence diagrams. However, the UML is more than just pretty pictures. The UML is a modeling meta-language implemented by software-design tools like Embarcadero Technologies' Describe and Sybase's PowerDesigner. Software engineers can use these design tools to control the complete object-oriented software engineering process. *Java For Artists* uses UML diagrams to illustrate program designs.

Physical Complexity

Physical complexity is that aspect of a software system determined by the number of design and production documents and other artifacts produced by software engineers during the project lifecycle. A small project will generally have fewer, if any, design documents than a large project. A small project will also have fewer source-code files than a large project. As with conceptual complexity, the steps taken to manage the physical complexity of small projects will prove inadequate for larger projects. However, there are some techniques you can learn and apply to small programming projects that you can in turn use to help manage the physical complexity of large projects as well.

THE RELATIONSHIP BETWEEN PHYSICAL AND CONCEPTUAL COMPLEXITY

Physical complexity is related to conceptual complexity in that the organization of a software system's architecture plays a direct role in the organization of a project's physical source-code files. A simple programming project consisting of a handful of classes might be grouped together in one directory. They might all be easily compiled by compiling every class in the directory at the same time. However, the same one-directory organization would simply not work on a large project with teams of programmers creating and maintaining hundreds or thousands of source files.

MANAGING PHYSICAL COMPLEXITY

You can manage physical complexity in a variety of ways. Selecting appropriate class names and package structures are two basic techniques that will prove useful not only for small projects, but for large projects as well. However, large projects usually need some sort of configuration-management tool to enable teams of programmers to work together on large source-code repositories. CVS and PVCS are two examples of configuration-management tools. The projects in this book do not require a configuration-management tool. However, the lessons you will learn regarding class naming and package structure can be applied to large projects as well.

MAXIMIZE COHESION – MINIMIZE COUPLING

An important way to manage both conceptual and physical complexity is to maximize software module cohesion and minimize software module coupling.

Cohesion is the degree to which a software module sticks to its intended purpose. A high degree of module cohesion is desirable. For example, a method intended to display an image on the screen would have high cohesion if that's all it did, and poor cohesion if it did some things unrelated to image display.

Coupling is the degree to which one software module depends on software modules external to itself. A low degree of coupling is desirable. Coupling can be controlled in object-oriented software by depending upon interfaces or abstract classes rather than upon concrete implementation classes. These concepts are explained in detail later in the book.

JAVA SOURCE FILE STRUCTURE

This section provides a brief introduction to the structure of a typical Java class, source file, and application. Example 1.1 gives the source code for a Java class named `SampleClass`.

This source code appears in a file named `SampleClass.java`. You can create the `SampleClass.java` file with a text editor such as NotePad, TextEdit, or with the text editor that comes built-in with your chosen integrated development environment (IDE). There can be many class definitions in one Java source file, but only one public class. The name of the file must match the name of the public class with the `.java` suffix added.

Line 1 gives a package directive stating that `SampleClass` belongs to the `com.pulpfreepress.jfa.chapter1` package. The package directive, if one exists, must be the first non-comment line in a Java source file.

Line 3 is an import directive. An import directive does not physically import anything into the Java source file. Rather, it allows programmers to use shortcut names for components belonging to included package names.

The `SampleClass` class declaration starts on line 5 and ends on line 33. Everything between the opening brace “{” on line 5 and the closing brace “}” on line 33 is in the body of the class definition.

A multi-line comment appears on lines 6 through 8. Java allows three types of comments, and each type is discussed in greater detail later in this chapter.

Class and instance field declarations appear on lines 9 through 11. The class-wide constant named `CONST_VAL` is declared on line 9 using the keywords `static` and `final`. A class-wide variable named `class_variable` is declared on line 10. Notice the difference between the names `CONST_VAL` and `class_variable`. `CONST_VAL` is in uppercase letters with an underscore character separating each word. It is generally considered to be good programming practice to put constant identifiers in uppercase letters to distinguish them from variables which appear in lowercase.

1.1 SampleClass.java

```

1  package com.pulpfreepress.jfa.chapter1; ← package declaration
2
3  import java.util.*; ← import statement
4
5  public class SampleClass { ← class definition start
6      /*****
7          Class and instance field declarations ← comment block
8          *****/
9      public static final int CONST_VAL = 25;
10     private static int class_variable = 0;
11     private int instance_variable = 0;
12
13     public SampleClass() { ← constructor
14         System.out.println("Sample Class Lives!");
15     }
16
17     public static void setClassVariable(int val) {
18         class_variable = val;
19     }
20
21     public static int getClassVariable() {
22         return class_variable;
23     }
24
25     public void setInstanceVariable(int val) {
26         instance_variable = val;
27     }
28
29     public int getInstanceVariable() {
30         return instance_variable;
31     }
32 } ← class definition end
33

```

An instance variable named `instance_variable` is declared on line 11. Every instance of `SampleClass` will have its own copy of `instance_variable`, but share a copy of `class_variable`.

A special method named `SampleClass()` is defined on line 13. The `SampleClass()` method is referred to as a *constructor method*. The constructor method is special because it bears the same name as the class in which it appears and has no return type. In this case, the `SampleClass()` constructor method prints a simple message to the console when instances of `SampleClass` are created.

A static method named `setClassVariable()` is defined beginning on line 17. The `setClassVariable()` method takes an integer argument and uses it to set the value of `class_variable`. Static methods are known as *class methods*, whereas non-static methods are referred to as *instance methods*. I will show you the difference between these two method types later in this chapter.

Another static method named `getClassVariable()` is declared and defined on line 21. The `getClassVariable()` method takes no arguments and returns an integer value. In this case, the value returned is the value of `class_variable`.

The last two methods, `setInstanceVariable()` and `getInstanceVariable()` are non-static instance methods. They work like the previous methods but only on instances of `SampleClass`.

SAMPLECLASS IN ACTION

Although `SampleClass` is a complete Java class, it cannot be executed by the Java Virtual Machine. A special type of class known as an application must first be created. Example 1.2 gives the source code for a class named `ApplicationClass`.

`ApplicationClass.java` is similar to `SampleClass.java` in that it contains a package-declaration statement as the first non-comment line. It also contains a class declaration for `ApplicationClass`, which is the same name as the Java source file. The primary difference between `SampleClass` and `ApplicationClass` is that `ApplicationClass` has a special method named `main()`. A Java class that contains a `main()` method is an application.

The `main()` method is declared on line 5. It takes an array of strings as an argument although in this example this feature is not used.

1.2 ApplicationClass.java

```

1  package com.pulpfreepress.jfa.chapter1;
2
3  public class ApplicationClass { ←————— class definition start
4
5      public static void main(String args[]){ ←————— main() method start
6          SampleClass sc = new SampleClass();
7          System.out.println(SampleClass.CONST_VAL);
8          System.out.println(SampleClass.getClassVariable());
9          System.out.println(sc.getInstanceVariable());
10         SampleClass.setClassVariable(3);
11         sc.setInstanceVariable(4);
12         System.out.println(SampleClass.getClassVariable());
13         System.out.println(sc.getInstanceVariable());
14         System.out.println(sc.getClassVariable());
15     }
16 }

```

An instance of `SampleClass` named `sc` is declared and created on line 6. Then, from lines 7 through 14, both the `SampleClass` and the instance of `SampleClass`, `sc`, are used in various ways to illustrate how to invoke each type of method (*static and instance*).

To test these classes, they first must be compiled with the `javac` compiler and then run using the `java` command. The results of running example 1.2 are shown in figure 1-1. For detailed discussions regarding how to compile these Java classes using various development environments refer to chapter 2.

```

Terminal - rch (ttypl)
[Rick-Millers-Computer:~/desktop/java_jars] swodog% java -jar ApplicationClass.jar
Sample Class Lives!
25
0
0
3
4
3
[Rick-Millers-Computer:~/desktop/java_jars] swodog%

```

Figure 1-2: Results of Running Example 1.2

GENERAL RULES FOR CREATING JAVA SOURCE FILES

There are a few rules you must follow when creating Java source files, and you were introduced to a few of them above.

First, Java source files contain plain text (ASCII), so use a plain-text editor to create the source file. (*You could use a word processing program like Microsoft Word to create a Java source file but you would have to save the file as plain text.*) The characters appearing in a source file can be UNICODE characters. But since most text editors only provide support for ASCII characters, Java programs can be written in ASCII. Escape sequences can be used to include ASCII representations of UNICODE characters if they are required in a program.

Second, a Java source file can contain an optional package directive. If a package directive appears in a source file it must be the first non-comment line in that source file.

Third, a Java source file can contain zero or more import directives. Import directives do not physically import anything into a Java source file; rather, they provide component-name shortcuts. For example, to use a Swing component in a Java program you could do one of two things: 1) you could include the “`import javax.swing.*;`” import directive in your source file and then use individual component names such as `JButton` or `JTextField` in your program as necessary, or, 2) you could leave out the import directive and use the fully qualified name for each Swing component you need, such as “`javax.swing.JButton`” or “`javax.swing.JTextField`”.

Fourth, a Java source file can contain any number of top-level class or interface definitions, but there can only be one public top-level class or interface in the source file. The name of the source file must match the name of this public class or interface with the addition of the `.java` file extension.

Finally, each top-level class can contain any number of publicly declared inner or nested class definitions. You will soon learn, however, it is not always a good idea to use nested or inner classes. Table 1-1 summarizes these Java source-file rules:

Issue	Rule
<i>1. Java source file creation and character composition</i>	The UNICODE character set can be used to create a Java source file, however, since most text editors support only the ASCII character set, source files are created in ASCII and UNICODE characters represented when necessary using ASCII escape sequences. A source file must end in a .java file extension.
<i>2. Package directive</i>	Optional — If a package directive appears in a source file it must be the first non-comment line in the file.
<i>3. Import directives</i>	Optional — There can be zero or more import directives. Import directives allow you to use unqualified component names in you programs.
<i>4. Top-level class and interface definitions</i>	There can be one or more top-level class or interface definitions in a source file, but, there can only be one public class or interface in a source file. The filename must match the name of the public class or interface. A source file must end with a .java file extension.
<i>5. Nested and inner classes</i>	A top-level class declaration can contain any number of nested or inner class definitions. <i>(I recommend avoiding nested and inner classes whenever possible.)</i>

Table 1-1: Java Source File Rules Summary

RULE-OF-THUMB: ONE CLASS PER FILE

Although Java allows multiple classes and interfaces to be defined in a single source file, it's helpful to limit the number of classes or interfaces to one per file. One class per file helps you manage a project's physical complexity because finding your source files when you need them is easy when the name of the file reflects the name of the class or interface it contains.

AVOID ANONYMOUS, NESTED, AND INNER CLASSES

Java also allows anonymous, nested, and inner classes to be defined within a top-level class. Anonymous, nested, and inner class definitions add a significant amount of conceptual complexity if used willy-nilly. Their use can also complicate the management of physical complexity, because they can be hard to locate when necessary.

CREATE A SEPARATE MAIN APPLICATION FILE

A Java class can contain a `main()` method. The presence of a `main()` method changes an otherwise ordinary class into an application that can be run by the Java Virtual Machine. Novice students, when writing their first Java programs, are often confused by the presence of the `main()` method. Therefore, when writing Java programs, I recommend you create a small class whose only purpose is to host the `main()` method and serve as the entry point into your Java program. Refer to examples 1.1 and 1.2 for examples of this approach.

PACKAGES

Packages provide a grouping for related Java classes, interfaces, and other reference types. For example, the Java platform provides many helpful utility classes in the `java.util` package, input/output classes in the `java.io` package, networking classes in the `java.net` package, and lightweight GUI components in the `javax.swing`

package. Using packages is a great way to manage both conceptual and physical complexity. In fact, the package structure of large Java projects is dictated by an application's software architectural organization.

In *Java For Artists* I will show you how to work with packages. A package structure is simply a directory hierarchy. Related class and interface source-code files are placed in related directories. The ability to create your own package structures will prove invaluable as you attempt larger programming projects.

I will use several package naming conventions to organize the code in this book. For starters, I will place short, simple programs in the default package. By this I mean that the classes belonging to short demonstration programs, with the exception of the code presented in chapters 1 and 2, will reside in the default package. Now I know this is cheating because the default package is specified by the *absence* of a package declaration statement. If you omit a package declaration statement from your programs, they too will reside in the default package. I do this, honestly, because novice students find the concept of packages very confusing at first. Learning how to create and utilize home-grown packages is a source of frustration that can be safely postponed until students master a few basic Java skills.

I use the following package naming convention for the code presented in chapters 1 and 2:

```
com.pulpfreepress.jfa.chapter1
```

An example of this package structure appeared earlier in examples 1.1 and 1.2. For complex projects presented in later chapters I slacked off a bit and shortened the package naming convention to the following:

```
com.pulpfreepress.package_name
```

Both of these package-naming conventions follow Sun's package-naming recommendations. I recommend that you use the following package naming convention for your projects:

```
lastname.firstname.project_name
```

So, for example, if you created a project named RobotRat and your name was Rick Miller, the code for your project would reside in the following package structure:

```
miller.rick.RobotRat
```

COMMENTING

The Java programming language provides three different ways to add comments to your source code. Using comments is a great way to add documentation directly to your code, and you can place them anywhere in a source-code file. The javac compiler ignores all three types of comments when it compiles the source code. However, special comments known as javadoc comments can be used by the javadoc tool to create professional-quality source code documentation for your programs. All three types of comments are discussed in detail in this section.

SINGLE-LINE COMMENTS

You can add single-line comments to Java programs using the characters “//” as shown below:

```
// This is an example of a single-line comment
```

The compiler ignores everything appearing to the right of the double back slashes up to the end of line.

MULTI-LINE COMMENTS

Add multi-line comments to Java programs using a combination of “/*” and “*/” character sequences as shown here:

```
1      /* This is an example of a multi-line comment. The compiler ignores
2      * everything appearing between the first slash-asterisk combination
3      * and the next asterisk-slash combination. It is often helpful to end
4      * the multi-line comment with the asterisk-slash sequence aligned to
5      * the left as shown on the next line.
6      */
```

JAVADOC COMMENTS

Javadoc comments are special multi-line comments that are processed by javadoc, the Java documentation generator. The javadoc tool automatically creates HTML-formatted application programming interface (API) documentation based on the information gleaned from processing your source files, plus any information contained in javadoc comments.

Javadoc comments begin with the characters “/” and end with the characters “*/”. Javadoc comments can contain descriptive paragraphs, doc-comment tags, and HTML markup. Example 1.3 shows a file named TestClass.java that contains embedded javadoc comments.

1.3 TestClass.java

```

1      /*****
2      TestClass demonstrates the use of <b>javadoc</b> comments.
3      @author Rick Miller
4      @version 1.0, 09/20/03
5      *****/
6      public class TestClass {
7
8          private int its_value;
9
10         /**
11          * TestClass constructor
12          * @param value An integer value used to set its_value
13          */
14         public TestClass(int value){
15             its_value = value;
16         }
17
18         /**
19          *getValue method
20          * @return integer value of its_value
21          */
22         public int getValue(){
23             return its_value;
24         }
25
26         /**
27          * setValue method
28          * @param value Used to set its_value
29          */
30         public void setValue(int value){
31             its_value = value;
32         }
33     }

```

GENERATING JAVADOC EXAMPLE OUTPUT

Figure 1-2 shows the javadoc tool being used in the UNIX environment to create API documentation for the TestClass.java file:

Figure 1-3 shows the results of using the javadoc tool to process the TestClass.java file:

For a complete reference to the javadoc tool and the different doc-comment tags available for use, consult either the java.sun.com web site or the *Java in a Nutshell* book listed in the references section at the end of this chapter.

IDENTIFIER NAMING - WRITING SELF-COMMENTING CODE

Self-commenting code is an identifier-naming technique you can use to effectively manage both physical and conceptual complexity. An identifier is a sequence of Java characters or digits used to form the names of entities used in your program. Examples of program entities include classes, constants, variables, and methods. All you have to do to write self-commenting code is to 1) give meaningful names to your program entities, and 2) adopt a consistent identifier-naming convention. Java allows unlimited-length identifier names, so you can afford to be descriptive.

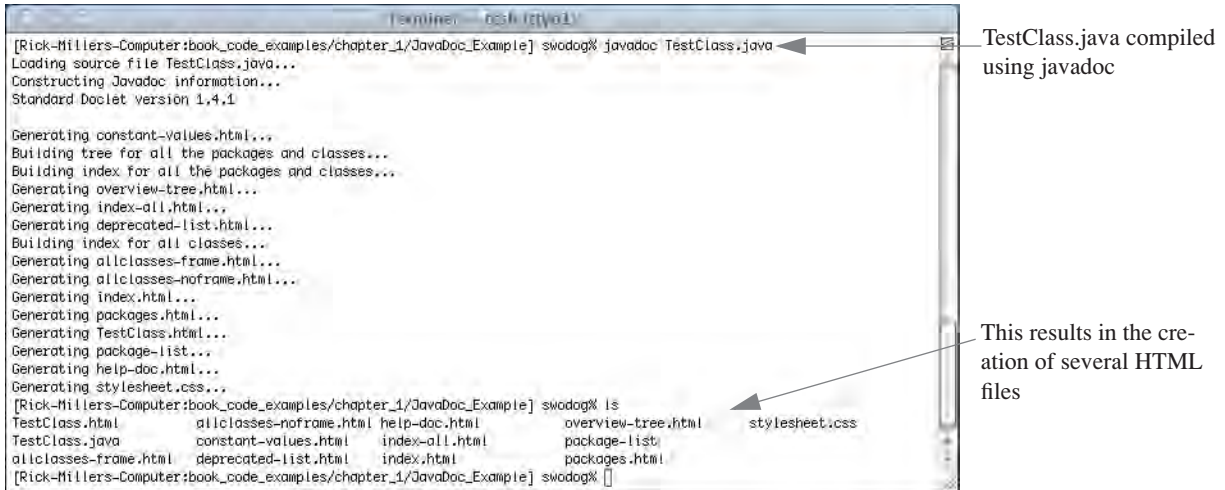


Figure 1-3: javadoc Tool Being Used to Generate TestClass API Documentation



Figure 1-4: Example HTML Documentation Page Created With javadoc

BENEFITS OF SELF-COMMENTING CODE

The benefits of using self-commenting code are many. First, self-commenting code is easier to read. Code that's easy to read is easy to understand. If your code is easy to read and understand, you will spend much less time tracking down logic errors or just plain mistakes.

CODING CONVENTION

Self-commenting code is not just for students. Professional programmers (*real professionals, not the cowboys!*) write self-commenting code because their code is subject to peer review. To ensure all members of a programming

team can read and understand each other's code, the team adopts a coding convention. The coding convention specifies how to form entity names, along with how the code must be formatted.

When you write programs to satisfy the exercises in *Java For Artists* I recommend you adopt the following identifier naming conventions:

CLASS NAMES

Class names should start with an initial capital letter. If the class name contains multiple words, then capitalize the first letter of each subsequent word used to form the class name. Table 1-2 offers several examples of valid class names:

Class Name	Comment
Student	One-syllable class name. First letter capitalized.
Engine	Another one-syllable class name.
EngineController	Two-syllable class name. First letter of each word capitalized.
HighOutputEngine	Three-syllable class name. First letter of each word capitalized.

Table 1-2: Class Naming Examples

CONSTANT NAMES

Constants represent values in your code that cannot be changed once initialized. Constant names should describe the values they contain, and should consist of all capital letters to set them apart from variables. Connect each word of a multiple-word constant by an underscore character. Table 1-3 gives several examples of constant names:

Constant Name	Comment
PI	Single-word constant in all caps. Could be the constant value of π .
MAX	Another single-word constant, but max what?
MAX_STUDENTS	Multiple-word constant separated by an underscore character.
MINIMUM_ENROLLMENT	Another multiple-word constant.

Table 1-3: Constant Naming Examples

VARIABLE NAMES

Variables represent values in your code that can change while your program is running. Variable names should be formed from lower-case characters to set them apart from constants. Variable names should describe the values they contain. Table 1-4 shows a few examples of variable names:

Variable Name	Comment
size	Single-word variable in lower-case characters. But size of what?
array_size	Multiple-word variable, each word joined by underscore character.
current_row	Another multiple-word variable.
mother_in_law_count	Multiple-word variable, each word joined by an underscore character.

Table 1-4: Variable Naming Examples

Method Names

A method represents a named series of Java statements that perform some action when called in a program. Since methods invoke actions their names should be formed from action words (*verbs*) that describe what they do. Begin method names with a lower-case character, and capitalize each subsequent word of a multiple-word method. The only exception to this naming rule is class-constructor methods, which must be exactly the same name as the class in which they appear.

Method Name	Comment
<code>printFloor</code>	Multiple-word method name. First letter of first word is lower-case; first letter of second word is upper-case. The first word is an action word.
<code>setMaxValue</code>	Multiple-word method name. This is an example of a mutator method.
<code>getMaxValue</code>	Another multiple-word method name. This is an example of an accessor method.
<code>start</code>	A single-word method name.

Table 1-5: Method Naming Examples

SUMMARY

The source of a student's difficulty with learning a programming language lies not with the language itself, but with the many other skills that must be mastered almost simultaneously along the way. You will find it helpful to know the development roles you must play and to have a project-approach strategy.

The three development roles you will play as a student are those of analyst, architect, and programmer. As the analyst, strive to understand the project's requirements and what must be done to satisfy those requirements. As the architect, you are responsible for the design of your project. As the programmer, you will implement your project's design in the Java programming language.

The project-approach strategy helps both novice and experienced students systematically formulate solutions to programming projects. The strategy deals with the following areas of concern: application requirements, problem domain, language features, and application design. By approaching projects in a systematic way, you can put yourself in control and can maintain a sense of forward momentum during the execution of your projects. The project-approach strategy can also be tailored to suit individual needs.

Programming is an art. Formulating solutions to complex projects requires lots of creativity. There are certain steps you can take to stimulate your creative energy. Sketch the project design before sitting at the computer. Reserve quiet space in which to work and, if possible, have a computer dedicated to school and programming projects.

There are five steps to the programming cycle: plan, code, test, integrate, and refactor.

Use method stubbing to test sections of source code without having to code the entire method.

There are two types of complexity: conceptual and physical. Object-oriented programming and design techniques help manage conceptual complexity. Physical complexity is managed with smart project file-management techniques, by splitting projects into multiple files, and using packages to organize source code.

Self-commenting source code is easy to read and debug. Adopt smart variable, constant, and method-naming conventions and stick with them.

Maximize cohesion — minimize coupling!

Skill-Building Exercises

1. **Variable Naming Conventions:** Using the suggested naming convention for variables, derive a variable name for

each of the concepts listed below:

Number of oranges

Required waivers

Day of week

Month

People in line

Next person

Average age

Student grades

Final grade

Key word

2. **Constant Naming Conventions:** Using the suggested naming convention for constants, derive a constant name for each of the concepts listed below:

Maximum student count

Minimum employee pay

Voltage level

Required pressure

Maximum array size

Minimum course load

Carriage return

Line feed

Minimum lines

Home directory

3. **Method Naming Conventions:** Using the suggested naming convention for methods, derive a method name for each of the concepts listed below:

Sort employees by pay

List student grades

Clear screen

Run monthly report

Engage clutch

Check coolant temperature

Find file

Display course listings

Display menu

Start simulation

SUGGESTED PROJECTS

1. **Feng Shui:** If you haven't already done so, stake your claim to your own quiet, private space where you will work on your programming projects. If you are planning on using the school's programming lab, stop by and familiarize yourself with the surroundings.
2. **Procure and Install IDE:** If you are doing your programming on your own computer make sure you have procured and loaded an integrated development environment (IDE) that will meet your programming requirements. If in doubt, check with your instructor.
3. **Project-Approach Strategy Checklist:** Familiarize yourself with the project-approach strategy checklist in appendix A.
4. **Obtain Reference Books:** Seek your instructor's or a friend's recommendation of any Java reference books that might be helpful to you during this course. There are also many good computer book-review sites available on the Internet. Also, there are many excellent Java reference books listed in the reference section of each chapter in this book.
5. **Web Search:** Conduct a Web search for Java and object-oriented programming sites. Bookmark any site you feel might be helpful to you as you master the Java language.

SELF-TEST QUESTIONS

1. List at least seven skills you must master in your studies of the Java programming language.
2. What three development roles will you play as a student?
3. What is the purpose of the project-approach strategy?
4. List and describe the four areas of concern addressed in the project-approach strategy.
5. List and describe the five steps of the programming cycle.
6. What are the two types of complexity?
7. What is meant by the term *isomorphic mapping*?
8. Why do you think it would be helpful to write self-commenting source code?
9. What can you do in your source code to maximize cohesion?
10. What can you do in your source code to minimize coupling?

REFERENCES

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Massachusetts, 2000. ISBN 201-61641-6

James Gosling, et. al. *The Java Language Specification, Second Edition*. Addison-Wesley, Boston, Massachusetts. ISBN: 0-201-31008-2

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*. Fourth Edition. O'Reilly & Associates, Sebastopol, CA. ISBN: 0-596-00283-1

Daniel Goleman. *Emotional Intelligence: Why it can matter more than IQ*. Bantam Books, New York, NY. ISBN: 0-553-37506-7

NOTES

CHAPTER 2



SUN CHAIRS

SMALL VICTORIES: CREATING JAVA™ PROJECTS

LEARNING OBJECTIVES

- *LIST AND DESCRIBE THE STEPS REQUIRED TO CREATE AND RUN A JAVA™ PROJECT*
- *STATE THE PURPOSE AND USE OF AN INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)*
- *LIST AND DESCRIBE THE COMPONENTS OF A TYPICAL IDE*
- *DESCRIBE THE PURPOSE AND USE OF THE FOLLOWING JAVA COMMAND-LINE TOOLS: javac, java, javadoc, jar*
- *LIST THE STEPS REQUIRED TO CREATE A JAVA PROJECT USING MACINTOSH® OS® X DEVELOPER TOOLS*
- *LIST THE STEPS REQUIRED TO CREATE A JAVA PROJECT USING SUN'S JAVA 2 SOFTWARE DEVELOPMENT KIT™ (SDK)*
- *EXPLAIN IN WRITING HOW TO SET THE PATH AND CLASSPATH ENVIRONMENT VARIABLES*
- *LIST THE STEPS REQUIRED TO CREATE A JAVA PROJECT USING BORLAND'S JBuilder™ IDE*
- *LIST AND DESCRIBE THE STEPS REQUIRED TO CREATE AN EXECUTABLE JAR FILE*

INTRODUCTION

This chapter will get you up and running with your Java development environment of choice as quickly as possible. I call it “Small Victories” because you will experience a great feeling of progress when you get your Java development environment installed and you create, compile, and run your first program. You will literally leap out of your chair and do a victory dance when you’ve successfully put all the pieces together to get your first program to run!

The previous chapter presented an example of a typical Java source file and described, in general terms, the process required to create, compile, and run the program. This chapter takes that discussion even further and shows you in detail how to create a Java programming project using different types of development tools and operating systems. The operating system you use dictates the types of software-development tools available for your use. However, as you will soon see, all development tools share much in common, regardless of development platform.

I start the chapter with an overview of the Java-program creation, compilation, and execution process. It’s important to understand this process so you know the capabilities of your development environment. Next, I will discuss Java integrated development environments (IDEs) and show you how an IDE can speed up your development activities. I follow this with a discussion of how to set up your operating-system environment so your Java tools work correctly. Operating-system configuration issues seem overwhelmingly complicated to computer novices. That’s why I take the time to show you how to set your PATH and CLASSPATH environment variables in Windows 2000/XP and the Linux operating systems. Macintosh OS X users have it easy since the Java platform and its development tools come pre-installed.

Next, I will show you how to create, compile, and run Java projects using Sun’s Java 2 Software Development Kit (J2SDK) for Microsoft Windows. This includes a discussion of how to download and install the J2SDK. I will then show you how to create Java projects using UNIX and Mac OS X development tools and Borland’s JBuilder, a popular IDE. I will then show you how to create and run executable Java .jar files.

JAVA PROJECT CREATION PROCESS

Figure 2-1 gives an overview of the Java project creation process.

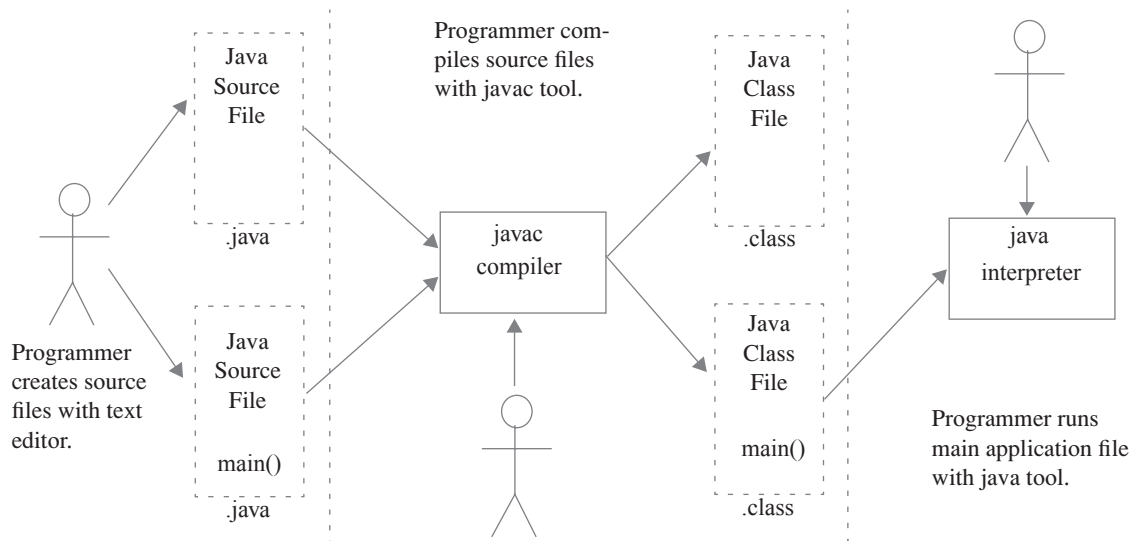


Figure 2-1: Java Project Creation Process

As figure 2-1 suggests, the Java project-creation process consists of three stages: source-file creation stage, source-file compilation stage, and main-application execution stage. Each stage is covered in detail in the following sections.

SOURCE FILE CREATION STAGE

In the source-file creation stage, a programmer creates Java source-code files using a text editor. A text editor is different from a word processor, although a word processor can be used to create Java source-code files. If you use a word processor to create a source-code file you must remember to save the file in plain text format. (*If you use Microsoft Word, make sure it does not append the .txt file extension to your source file. The java compiler will not be able to compile the file if it has a .txt file extension.*)

Java source code can consist of UNICODE characters. A UNICODE character is 16 bits long thus giving it the capability to display a large variety of language characters and symbols such as Korean, Chinese, or Japanese. However, since most text editors can only process ASCII characters, which are 8-bits long, you can represent UNICODE characters in ASCII using escape sequences. To include a UNICODE character using an escape sequence you must consult a UNICODE table to get the encoding for the character you wish to use. For more information about UNICODE you can visit [www.unicode.org/unicode/standard/standard.html].

Java source-code files must be saved with a .java file extension. The name of the file must match the class name of the public top-level class defined in the file. (*Note: There can be many classes defined within one source file, but only one public top-level class. A top-level class is simply a class that is not nested within another class. In the previous chapter, however, I recommended against placing multiple classes in one source file.*)

SOURCE FILE COMPILATION STAGE

The second stage of the Java project creation process is source-file compilation. The Java source-code files created in the previous stage must be transformed into byte code using the *javac* compiler tool. This results in one or more class files being created for each source file. A Java class file has a .class file extension. (*Note: If a source code file contains multiple class definitions and/or nested classes, then compilation will result in multiple classes, one for the top-level class and one for each nested class contained within a top-level class.*)

WHAT IF A SOURCE FILE CONTAINS ERRORS?

A source-code file that contains errors will not compile. In this case, the *javac* compiler tool will print a series of error messages to the screen. The number of errors in your code determines the number of error messages printed to the screen. If your source file contains errors and does not compile, you must return to the first stage and edit the file to correct the bugs. Note here that the compiler will only catch language syntax and semantic errors along with checked exceptions. (*Exceptions are covered in chapter 15.*) It will not catch errors in programming logic or fix bad algorithms.

When fixing compiler errors, remember the advice offered in chapter 1: fix the first compiler error first, otherwise you will only make matters worse.

MAIN APPLICATION EXECUTION STAGE

After the source files have been created and compiled, you must use the *java* interpreter to execute the class file containing the *main()* method. (*A class containing a *main()* method is referred to as an application class.*) Any class files the main application class depends upon must be located in the operating system's CLASSPATH. If, when you try to run a Java program, you get an error that says something like "class not found..." then you most likely have not properly configured your system's CLASSPATH environment variable.

AUTOMATING THE JAVA PROJECT CREATION PROCESS

The second and third stages of the project-creation process can be automated to a certain extent by various development tools. The next section talks about integrated development environments (IDEs) and shows you how an IDE can help speed up your development activities.

INTEGRATED DEVELOPMENT ENVIRONMENTS

A Java integrated development environment (IDE) is a set of integrated programming tools designed to work together to dramatically increase programmer efficiency. A basic IDE will combine source-file creation and editing, project-management, application-launch, and debugging services. Having one tool that combines these services alone is a big help. Full-featured IDEs can even assist with graphical user interface (GUI) layout and code generation. This section provides a very brief look at several IDEs. The tools discussed below are by no means the only IDEs available, but they do represent the best-of-breed.

TEXTPAD™ – J2SDK COMBINATION

An example of a simple IDE is the combination of the programmer's text editor TextPad with Sun's Java 2 Software Development Kit. TextPad integrates with the Java command-line tools so you can call the compiler and interpreter from within TextPad. Source-file compilation errors open in a TextPad window. With a few clicks, TextPad takes you to the offending line in the code where the error occurred. TextPad is included on the *Java For Artists* supplemental CD. You can also obtain TextPad from [www.textpad.com].

SUN'S NETBEANS™

Sun provides a full-featured IDE called NetBeans. NetBeans supports source-file creation, compiling, testing, and debugging of both Java 2 Standard Edition (J2SE) applications and Java 2 Enterprise Edition (J2EE) Web applications. It includes a full-featured text editor with syntax highlighting and error checking, visual design tools, Ant support, configuration-management system support, and many other features. NetBeans can be downloaded along with the J2SDK from Sun's Java website [www.java.sun.com].

BORLAND® JBuilder®

Borland's JBuilder is an example of an advanced IDE that's packed with features. Highlights include integration with CVS or PVCS version-control systems, embedded Java API documentation help and GUI layout tools. Borland provides a reduced-feature version of JBuilder called Foundation free from their website [www.borland.com]. If you have a slow Internet connection that limits your ability to download large software files, you can buy JBuilder Foundation on CD for a small fee. Advanced versions of JBuilder are not free, but the additional features they provide are considered by large numbers of Java developers across the world as being well worth the money spent.

Eclipse®

Many professional Java developers use an extremely powerful IDE called Eclipse. Developed by IBM, Eclipse is available as an open-source application from eclipse.org [www.eclipse.org]. Eclipse is too powerful to be described fully in this section, but imagine a development environment that can do just about anything you can think of and you have Eclipse. Eclipse gets its power from the plethora of open-source and commercial plug-ins available for use with the tool.

A side benefit to learning Eclipse would be your ability to segue easily to using IBM's flagship development environment called Websphere Studio Application/Site Developer. For more information about Eclipse and its available plug-ins, visit the Eclipse website.

CREATING JAVA PROJECTS USING MICROSOFT® WINDOWS® 2000/XP

This section shows you how to create Java projects using the Microsoft Windows operating system. However, success as a Java programmer requires more than just knowing how to click icons. You must learn to use the Windows Command Processor as well as how to configure your operating system for Java development.

Two important topics covered in this section include the setting of the PATH and CLASSPATH system variables. Once you've learned these concepts, you can safely and confidently let an IDE manage these issues for you.

SUN'S JAVA 2 STANDARD EDITION SOFTWARE DEVELOPMENT KIT (J2SDK)

The J2SDK, or simply the SDK, consists of the *javac* compiler, the *java* interpreter, the *javadoc* documentation generator, and many other command-line tools, some of which this book covers later.

You will need to perform the following steps to create Java programs using the SDK:

- **Step 1:** Download and install the J2SDK
- **Step 2:** Configure the operating system's PATH variable
- **Step 3:** Configure the operating system's CLASSPATH variable
- **Step 4:** Select a suitable text editor
- **Step 5:** Create source files
- **Step 6:** Compile source files
- **Step 7:** Execute the main-application class file

STEP 1: DOWNLOAD AND INSTALL THE J2SDK

You must first install the SDK before you can use the command-line tools to compile and run your Java programs. The *Java For Artists* supplemental CD-ROM contains the SDK. You can also download the latest version of the J2SDK direct from SUN. [www.java.sun.com] Figure 2-2 shows a partial view of Sun's Java 2 Standard Edition version 1.4.2 download page. (*The screen capture used in figure 2-2 was taken in late 2004. The look and feel of the Sun web site will change over time, but the content will remain similar to that shown here.*)

Referring to figure 2-2 — take note of several important things shown on this web page. First, the SDK is available for many other operating systems besides Microsoft Windows. Second, you can download the NetBeans IDE along with the SDK. Third, you can download the Java Runtime Environment (JRE) by itself. The JRE consists of the Java platform class files required to run Java applications. Fourth, you can download the SDK by itself. The SDK includes all the Java command-line tools required to build Java applications. The SDK also includes the JRE.

For this section you need only download the SDK or get it from the CD-ROM. Once the SDK download is complete, you will need to install it. The download process downloads and places an executable file on your desktop or other designated download directory. To begin the installation process simply double-click this executable file.

You can customize the Java SDK installation with the Custom Setup dialog as shown in figure 2.3. However, until you gain experience as a Java programmer, I strongly advise you to accept the recommended default installation. This installs the SDK in a directory that reflects the name of the version of the SDK being installed. Figure 2-3 illustrates the installation of the 1.4.2_01 version of the SDK. The directory in which this SDK will be installed is named "j2sdk1.4.2_01". The installer selects the C: drive as the default installation drive. This results in the full path name to the SDK of "C:\j2sdk1.4.2_01". If you install a different version of the SDK, your installation directory will have a different name.

STEPS 2 & 3: SETTING MICROSOFT WINDOWS ENVIRONMENT VARIABLES

Before you can use the Java SDK, you must tell the Microsoft Windows operating system where to find the Java command-line tools (*javac*, *java*, etc.), where to find the classes associated with the Java Runtime Environment (JRE), and where to find the classes you create when you compile your own Java source files. You can address all of these issues by setting two important operating system environment variables: PATH and CLASSPATH.



Figure 2-2: Sun's Java Download Page

The default settings installs the SDK in a directory that reflects the name of the version of the SDK you are installing

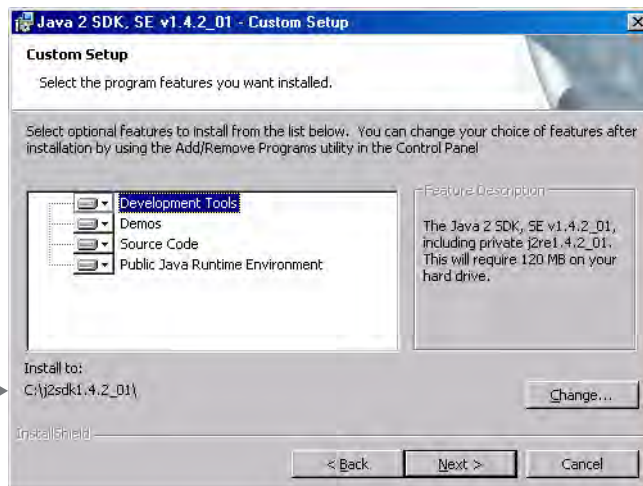


Figure 2-3: Custom Setup Dialog

PURPOSE OF PATH & CLASSPATH

The purpose of the PATH environment variable is to tell the operating system where to look for executable files. (An executable file has a .exe file extension.) The PATH environment variable is already set to some value by the operating system during initial installation, and perhaps modified later by additional software package installations. You must modify the PATH environment variable so that it knows where to look in the SDK directory for the Java command-line tools.

The purpose of the CLASSPATH environment variable is to tell the *java* interpreter where to look for Java class files. If this is the first time you are loading the Java SDK, or any Java-related tool on your computer, the CLASSPATH environment variable most likely does not yet exist.

SETTING THE SYSTEM PATH

You must modify the PATH environment variable's value to include the fully-qualified path to the directory that contains the Java command-line tools. This directory is named "bin" (*for binary*), and is a subdirectory of the jdk1.4.2_01 directory. Figure 2-4 shows the folders and files located in the jdk1.4.2_01 folder, including the bin folder. Figure 2-5 shows a listing of the bin directory contents

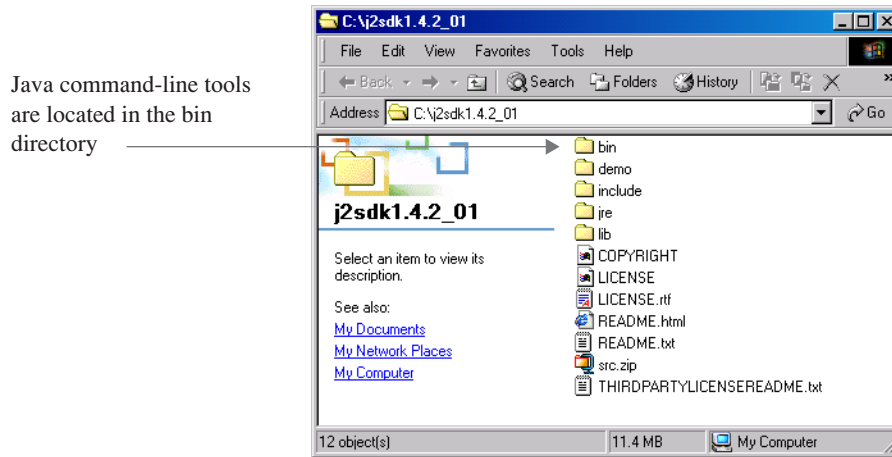


Figure 2-4: Window Showing Subfolders and Files in the j2sdk1.4.2_01 Folder

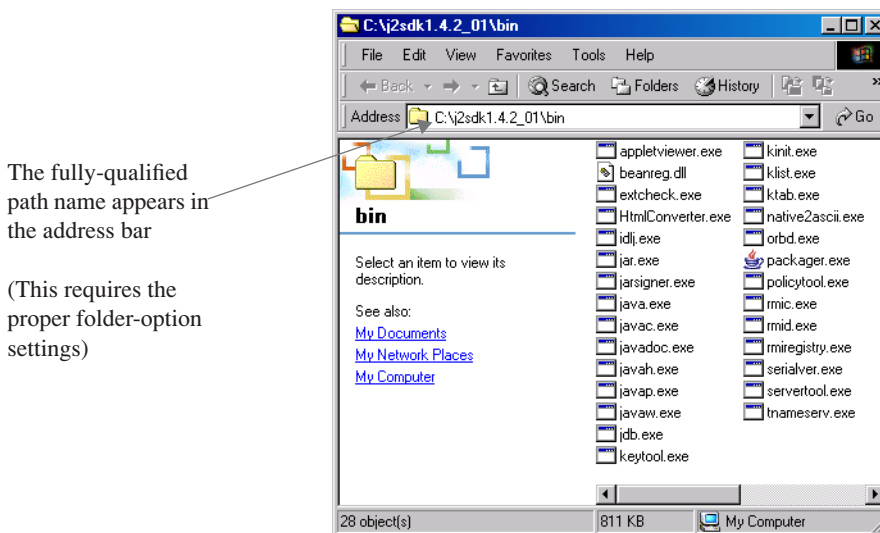


Figure 2-5: Contents Listing of C:\j2sdk1.4.2_01\bin

You will need the fully-qualified path name to the SDK bin directory when you set the PATH variable's value. The fully-qualified path name includes the drive letter, the name of the J2SDK directory, and the bin directory. For example, as is shown in figure 2-5, if you are using the J2SDK version 1.4.2_01 and it is installed on the C: drive, the fully-qualified path name to the bin directory will look like this:

```
C:\j2sdk1.4.2_01\bin
```


To set the PATH environment variable's value, right-click on the My Computer icon located on your desktop. This opens a pop-up menu. From the menu, click Properties to open the System Properties window. In the System Properties window, click the Advanced tab at the top right, then click the Environment Variables button located in the center of the tab window. This opens the Environment Variables window. Figure 2-6 shows the complete sequence described thus far.

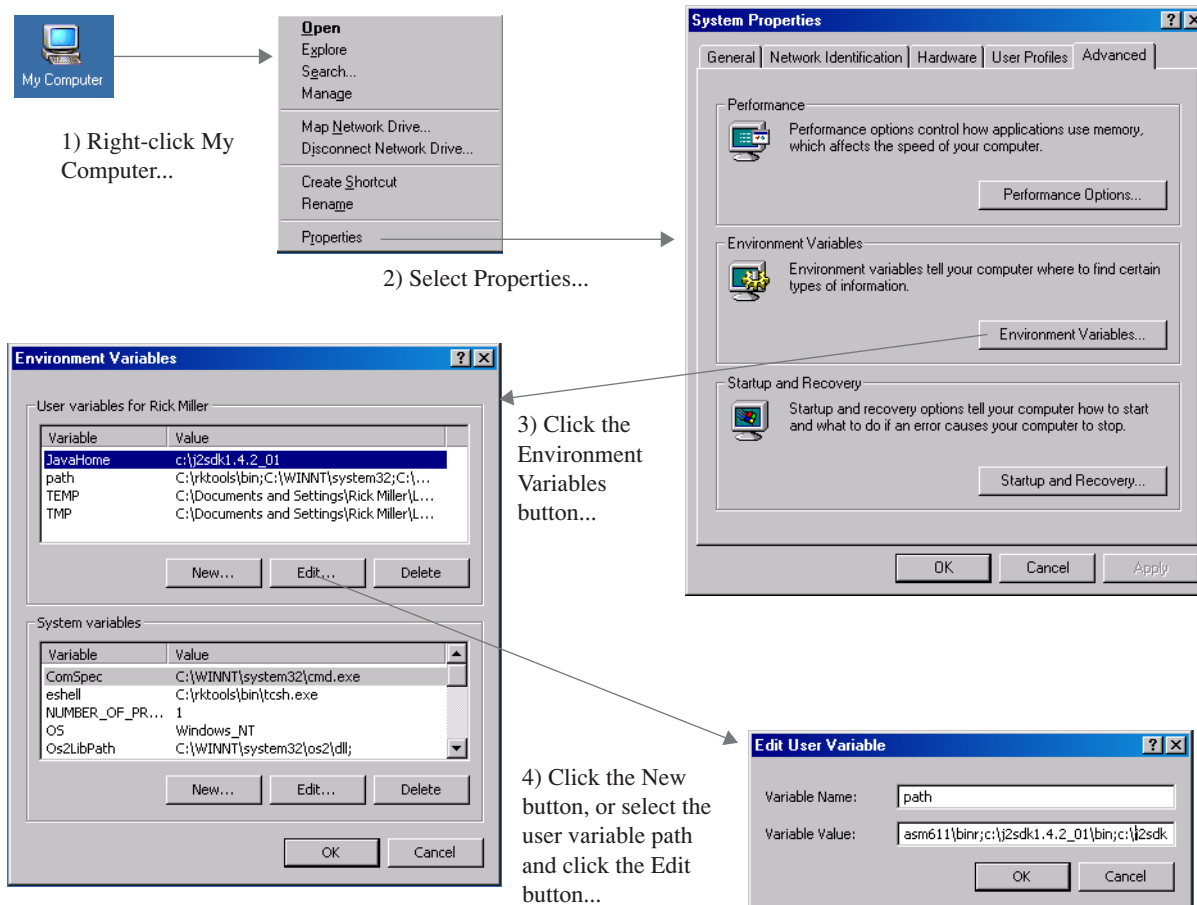


Figure 2-6: Setting the PATH Environment Variable in Microsoft Windows 2000/XP

When you open the Environment Variables window, you will see two sections. The upper part of the window shows the user variables. User variables apply to your user login. The lower part of the window shows the system variables. System variables apply to all users, but they can be augmented by user variables. System variables can only be modified by users having Administrator privileges.

There will be a system PATH variable, but I recommend leaving the system variable alone and creating a user PATH variable. If your user PATH variable does not already exist, click the New button located below the user variables section. In the Variable Name field enter the name of the user variable:

PATH

In the Variable Value field add the fully-qualified path value to the SDK bin directory:

C:\j2sdk1.4.2_01\bin;

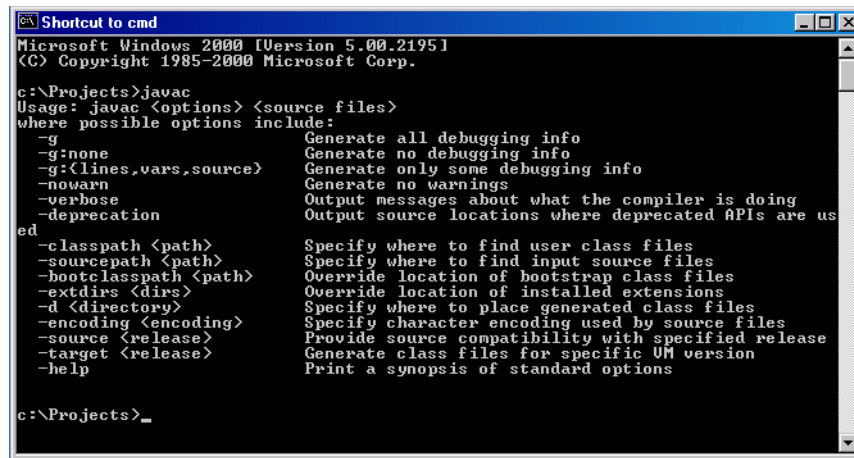
To this I would also add the path to the JRE bin directory. The complete PATH variable value will now resemble the following string:

C:\j2sdk1.4.2\bin;c:\j2sdk1.4.2_01\jre\bin;

Notice how the "C:\j2sdk1.4.2_01" portion of the path must be repeated for both the SDK and the JRE portion of the PATH variable value. Also notice how each separate path entry is separated by a semicolon.

When you are finished creating or editing the PATH environment variable, be sure to click the OK button on each window to save your changes.

It's now time to test the PATH environment variable. Open a command-processor window as shown in figure 2-7 and enter the command "javac". This should display the Java compiler help summary.



```

Shortcut to cmd
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

c:\Projects>javac
Usage: javac <options> <source files>
where possible options include:
  -g                Generate all debugging info
  -g:none           Generate no debugging info
  -g:{lines,vars,source}
                   Generate only some debugging info
  -nowarn           Generate no warnings
  -verbose          Output messages about what the compiler is doing
  -deprecation      Output source locations where deprecated APIs are used
-ed
  -classpath <path>    Specify where to find user class files
  -sourcepath <path>  Specify where to find input source files
  -bootclasspath <path>
                   Override location of bootstrap class files
  -extdirs <dirs>    Override location of installed extensions
  -d <directory>     Specify where to place generated class files
  -encoding <encoding>
                   Specify character encoding used by source files
  -source <release>  Provide source compatibility with specified release
  -target <release>  Generate class files for specific VM version
  -help             Print a synopsis of standard options

c:\Projects>_

```

Figure 2-7: Testing PATH Environment Variable by Typing javac

Now that you have set the PATH environment variable, the operating system can find the Java command-line tools. You must now tell the operating system where to find the Java Platform class files, as well as the class files you will create as part of your programming projects. You do this by setting the system CLASSPATH variable.

SETTING THE CLASSPATH ENVIRONMENT VARIABLE

You set the CLASSPATH environment variable the same way you set the PATH environment variable. In fact, you can follow the same steps shown in figure 2-6. However, before setting the CLASSPATH, you will need to know two things: 1) where the Java Platform class files are located, and 2) where you are going to put your class files when you do your program development.

Let's start with the location of the Java Platform class-file location. To run Java programs, you will need the JRE. The JRE was installed along with the SDK, and is located in the "jre" subdirectory of the SDK. In the jre subdirectory you will find another subdirectory named "lib" which stands for library. The lib directory contains the Java Platform Application Programming Interface (API) files required to run Java programs. The fully-qualified path name that should be used for the CLASSPATH environment variable will be similar to this:

```
C:\j2sdk1.4.2_01\jre\lib
```

Now, to tell the *java* interpreter to look in the current directory for your project's class files (*wherever you happen to be when you try running your Java programs*) add the following path value:

```
.;
```

Yes, it's a period followed by a semicolon. The full CLASSPATH variable value will now look like this:

```
C:\j2sdk1.4.2_01\jre\lib;.;
```

While you're at it, you may as well add one more search location to the CLASSPATH. Do this by adding the following path value:

```
.\classes;
```

This will instruct the JVM to search for classes in a directory named "classes", which is located in the working directory. The complete CLASSPATH variable value should now look like this:

```
C:\j2sdk1.4.2_01\jre\lib;. ; .\classes;
```

When you have finished setting the CLASSPATH variable click the OK buttons to save your changes. You should now have both your PATH and CLASSPATH environment variables set. It's time to test everything by creating a few classes, compiling them, and running the resulting application class file.

ANOTHER HELPFUL HINT: DISPLAY FILES SUFFIXES AND FULL PATH NAMES

Before you do another thing, open a folder and select Tools > Folder Options and click the Views tab at the top as shown in figure 2-8. These settings will turn Microsoft Windows into a novice-Java-programmer-friendly environment. When you are finished setting the folder options, click the OK button to apply the changes.

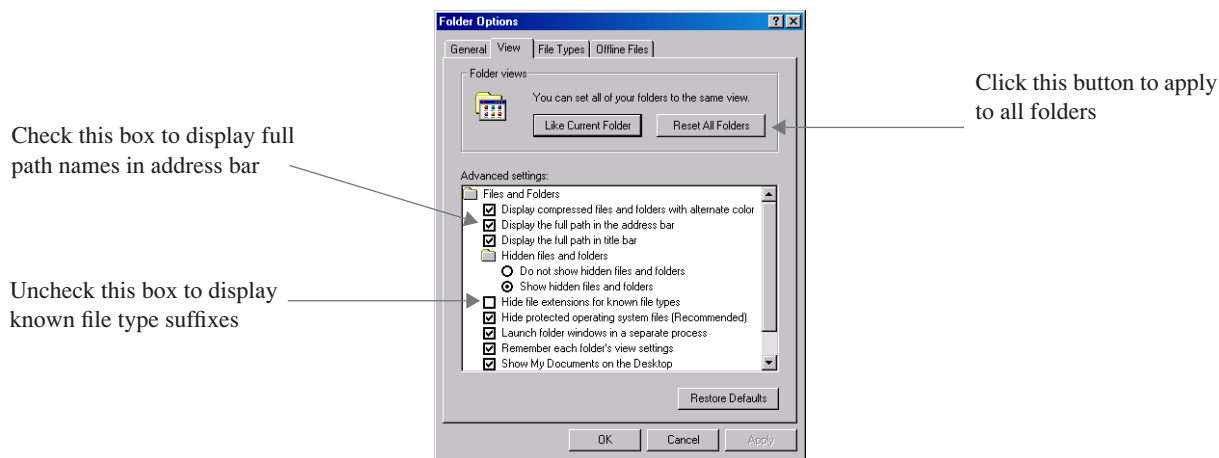


Figure 2-8: Setting Some Important Folder Options

STEPS 4 & 5: SELECT A TEXT EDITOR AND CREATE THE SOURCE FILES

Use a text editor of your choosing to create any Java source files you need for your project. To test your development environment, use the two source files SampleClass.java and ApplicationClass.java from chapter 1. They are given again below in examples 2.1 and 2.2. (Leave out the line numbers when copying code from the book.)

2.1 SampleClass.java

```

1      package com.pulpfreepress.jfa.chapter1;
2
3      import java.util.*;
4
5      public class SampleClass {
6          /*****
7              Class and instance field declarations
8              *****/
9          public static final int CONST_VAL = 25;
10         private static int class_variable = 0;
11         private int instance_variable = 0;
12
13         public SampleClass() {
14             System.out.println("Sample Class Lives!");
15         }
16
17         public static void setClassVariable(int val) {
18             class_variable = val;
19         }
20
21         public static int getClassVariable() {
22             return class_variable;
23         }
24
25         public void setInstanceVariable(int val) {
26             instance_variable = val;
27         }
28
29         public int getInstanceVariable() {
30             return instance_variable;
31         }
32     }

```

```

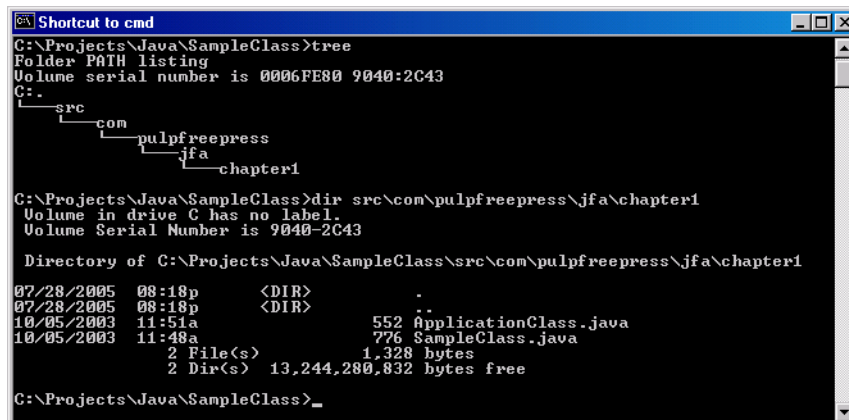
1     package com.pulpfreepress.jfa.chapter1;
2
3     public class ApplicationClass {
4
5         public static void main(String args[]){
6             SampleClass sc = new SampleClass();
7             System.out.println(SampleClass.CONST_VAL);
8             System.out.println(SampleClass.getClassVariable());
9             System.out.println(sc.getInstanceVariable());
10            SampleClass.setClassVariable(3);
11            sc.setInstanceVariable(4);
12            System.out.println(SampleClass.getClassVariable());
13            System.out.println(sc.getInstanceVariable());
14
15            System.out.println(sc.getClassVariable());
16        }
17    }

```

Notice how each of these source files belongs to the package named `com.pulpfreepress.jfa.chapter1`. I will leave that package name the same. When you create these files, you will want to put them in a directory somewhere on your hard drive. For the following examples I will use a directory with the following path name:

`C:\projects\Java\SampleClass`

The `SampleClass` directory will be the “*project directory*”. In the project directory you must create a subdirectory structure that reflects the package name specified in the source files. I also recommend that you place the package directory structure in a directory named “*src*” (which stands for source). Figure 2-9 shows the `tree` and `dir` commands being used to display the newly-created package structure and the two source files.



```

C:\Projects\Java\SampleClass>tree
Folder PATH listing
Volume serial number is 0006FE80 9040:2C43
C:
├── src
│   ├── com
│   │   ├── pulpfreepress
│   │   │   ├── jfa
│   │   │   │   └── chapter1

```

```

C:\Projects\Java\SampleClass>dir src\com\pulpfreepress\jfa\chapter1
Volume in drive C has no label.
Volume Serial Number is 9040-2C43

Directory of C:\Projects\Java\SampleClass\src\com\pulpfreepress\jfa\chapter1

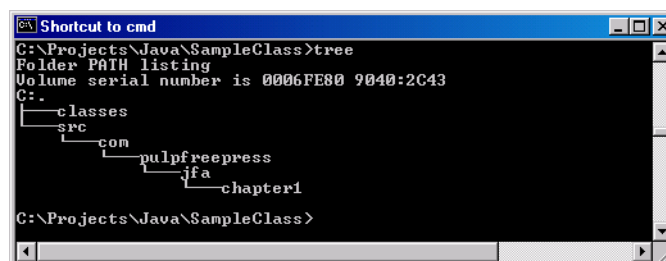
07/28/2005  08:18p    <DIR>          .
07/28/2005  08:18p    <DIR>          ..
10/05/2003  11:51a                552 ApplicationClass.java
10/05/2003  11:48a                776 SampleClass.java
                2 File(s)      1,328 bytes
                2 Dir(s)   13,244,280,832 bytes free

C:\Projects\Java\SampleClass>_

```

Figure 2-9: SampleClass Project Source-Code Directory Structure

Lastly, in the project directory, create one more subdirectory named “*classes*”. This directory will come in handy in the next step. Figure 2-10 shows the `tree` command being used, once again, on the `SampleClass` directory to reveal its final subdirectory structure.



```

C:\Projects\Java\SampleClass>tree
Folder PATH listing
Volume serial number is 0006FE80 9040:2C43
C:
├── classes
│   ├── src
│   │   ├── com
│   │   │   ├── pulpfreepress
│   │   │   │   ├── jfa
│   │   │   │   │   └── chapter1

```

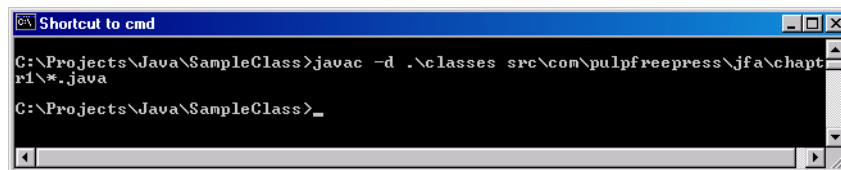
Figure 2-10: Final SampleClass Subdirectory Structure

STEP 6: COMPILING THE SOURCE FILES

When you've finished creating the `SampleClass.java` and `ApplicationClass.java` source files, and have placed them in the `chapter1` sub-directory of the source-file package structure, you are now ready to compile them with the `javac` compiler tool. From the `SampleClass` project directory issue the following command:

```
javac -d .\classes src\com\pulpfreepress\jfa\chapter1\*.java
```

Figure 2-11 shows the `javac` compiler command being used at the command prompt:



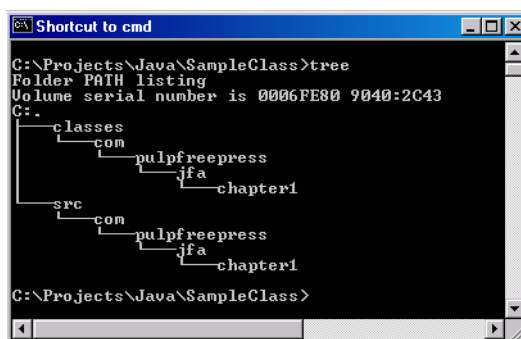
```

C:\Projects\Java\SampleClass>javac -d .\classes src\com\pulpfreepress\jfa\chapter1\*.java
C:\Projects\Java\SampleClass>_

```

Figure 2-11: Compiling the Java Source Files

If all goes well, and there are no compilation errors, the `javac` compiler will compile the source files, and place the resulting `.class` files in the `classes` directory, automatically creating the correct package structure. Figure 2-12 shows the `tree` command being used on the `SampleClass` directory after compilation.



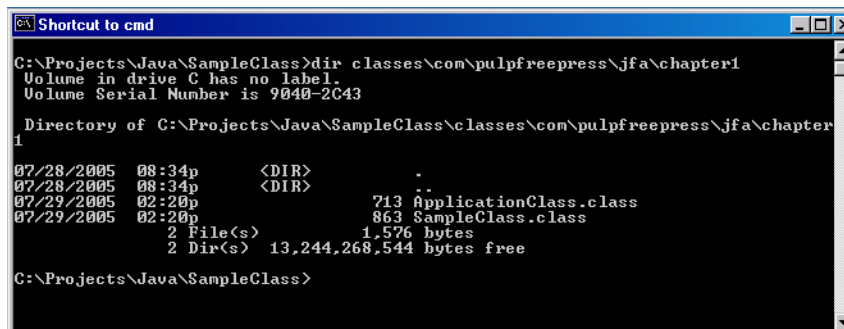
```

C:\Projects\Java\SampleClass>tree
Folder PATH listing
Volume serial number is 0006FE00 9040:2C43
C:
  classes
  src
    com
      pulpfreepress
        jfa
          chapter1

```

Figure 2-12: SampleClass Sub-directory Structure After Compilation

Figure 2-13 shows the `dir` command being used to display the `.class` files created as a result of the compilation process.



```

C:\Projects\Java\SampleClass>dir classes\com\pulpfreepress\jfa\chapter1
Volume in drive C has no label.
Volume Serial Number is 9040-2C43

Directory of C:\Projects\Java\SampleClass\classes\com\pulpfreepress\jfa\chapter1

07/28/2005  08:34p    <DIR>          .
07/28/2005  08:34p    <DIR>          ..
07/29/2005  02:20p                713 ApplicationClass.class
07/29/2005  02:20p                863 SampleClass.class
                2 File(s)          1,576 bytes
                2 Dir(s)   13,244,268,544 bytes free

C:\Projects\Java\SampleClass>

```

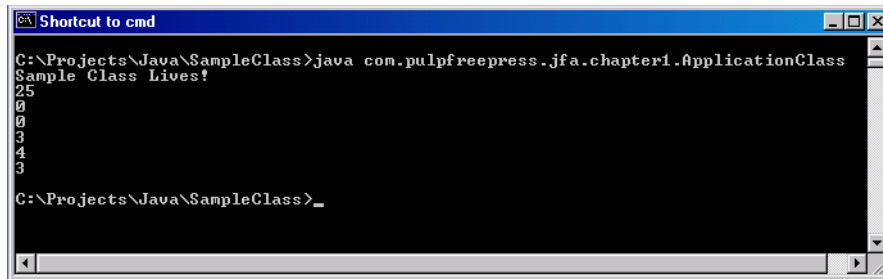
Figure 2-13: The `.class` Files are Located in Their Proper Package Structure

STEP 7: EXECUTING THE MAIN APPLICATION CLASS

You are now ready to run the program. Both source files have been compiled and the resulting class files have been automatically placed in the `com\pulpfreepress\jfa\chapter1` directory. From the command prompt enter the following command to run the program:

```
java com.pulpfreepress.jfa.chapter1.ApplicationClass
```

Figure 2-14 shows the results of running the `ApplicationClass` program.



```

C:\Projects\Java\SampleClass>java com.pulpfreepress.jfa.chapter1.ApplicationClass
Sample Class Lives!
25
0
0
3
4
3
C:\Projects\Java\SampleClass>_

```

Figure 2-14: Results of Running ApplicationClass Program

BORLAND'S JBuilder

JBuilder is an excellent example of a Java IDE. JBuilder integrates project-management features, source-code editing, debugging, GUI layout tools, and much, much more. When you use an IDE like JBuilder, you will notice an increase in your development efficiency. (*After you learn to use the tool of course!*)

Before you can effectively use JBuilder, however, you still must understand the purpose and use of the PATH and CLASSPATH environment variables and how to set them in the Windows operating system. Therefore, I strongly recommend mastering the techniques of creating Java projects with the SDK, as presented in the previous section, before moving to an IDE.

STEPS TO CREATING JAVA PROJECTS IN JBuilder

JBuilder, like any IDE, is project-centric. Once you understand this concept you can use just about any IDE, even though each may have a different look and feel and have slightly different features. The following general steps are required to create and run Java programs using JBuilder:

- **Step 1:** Create project
- **Step 2:** Create or add source files, packages, or other components
- **Step 3:** Set project properties if required
- **Step 4:** Identify the main application class
- **Step 5:** “Make” the project
- **Step 6:** Run the project

I discuss these steps in detail below as I show you how to create a JBuilder project with the SampleClass.java and ApplicationClass.java source files given in examples 2.1 and 2.2.

STEP 1: CREATE PROJECT

Open JBuilder and close any projects that may automatically open. The JBuilder window should look similar to that shown in figure 2-15. Next, select File > New Project as shown in figure 2-16. This will open the Project Wizard Step 1 of 3 dialog window as shown in figure 2-17.

The Project Wizard will guide you through the set up of your project. In step 1, select the project type, project name, project directory, and what template to use to create the project. For this example, I have selected the *jpx* project type and named the project “TestProject”. I have left JBuilder’s recommendation for the project location unchanged and accepted the default project as the template. Click the Next button to go to step 2.

Figure 2-18 shows step 2 of the Project Wizard. In this dialog window you set various project-path settings as well as any additional source-file locations, documentation, and shared libraries. For now, you can accept the default values recommended by JBuilder. Click the Next button to proceed to step 3.

Figure 2-19 shows step 3 of the Project Wizard. Here you can set the project encoding, package-exposure level, and default values for javadoc comments. You can accept all the JBuilder recommended defaults for this example. When you’ve finished step 3 of the Project Wizard click the Finish button.

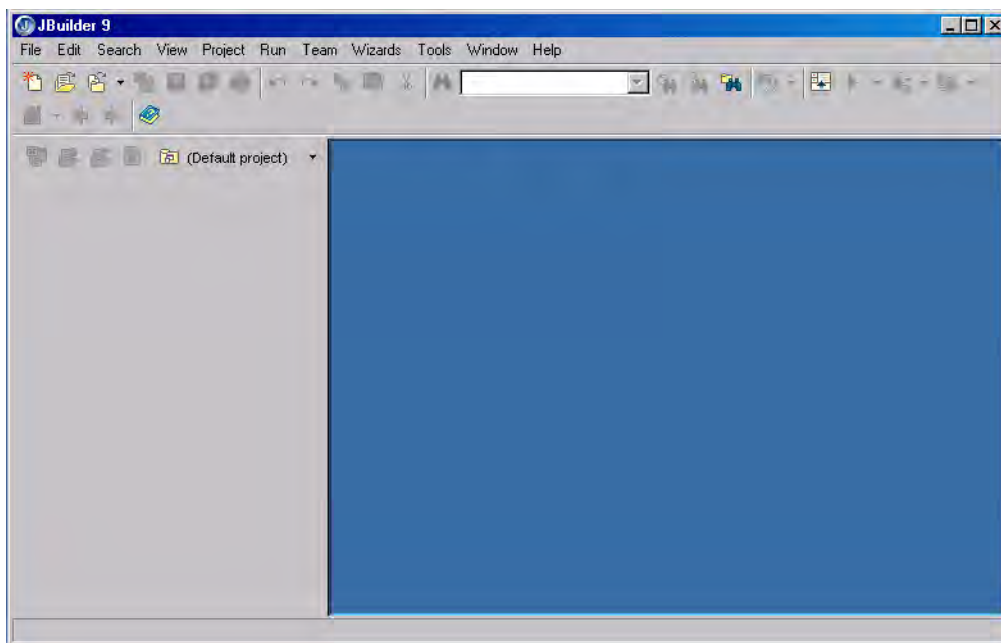


Figure 2-15: JBuilder with No Open Projects

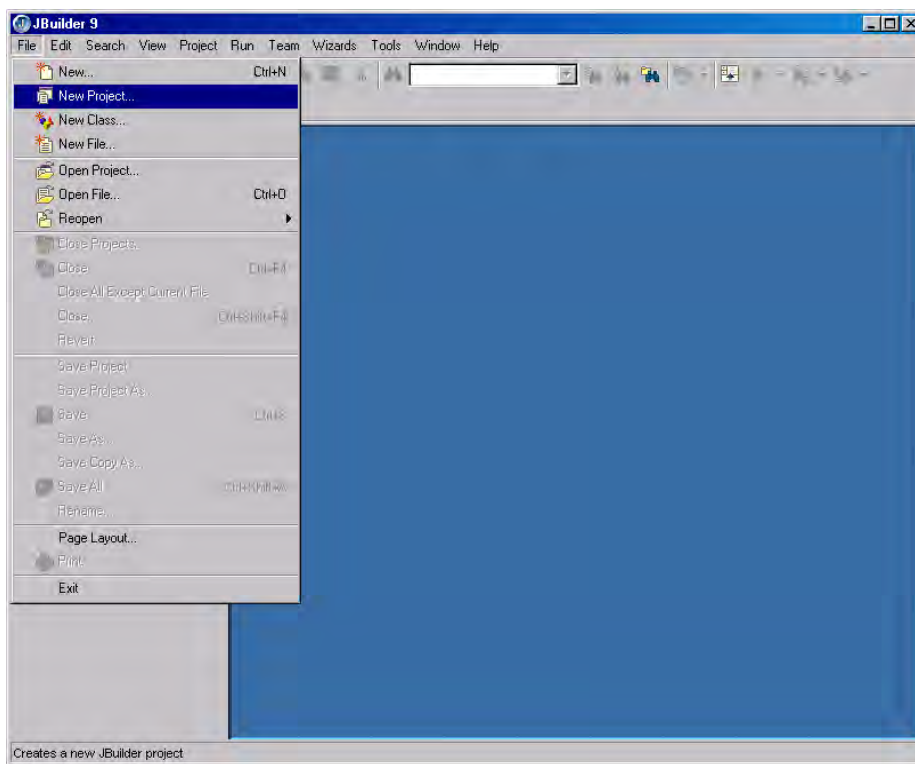


Figure 2-16: New Project Menu

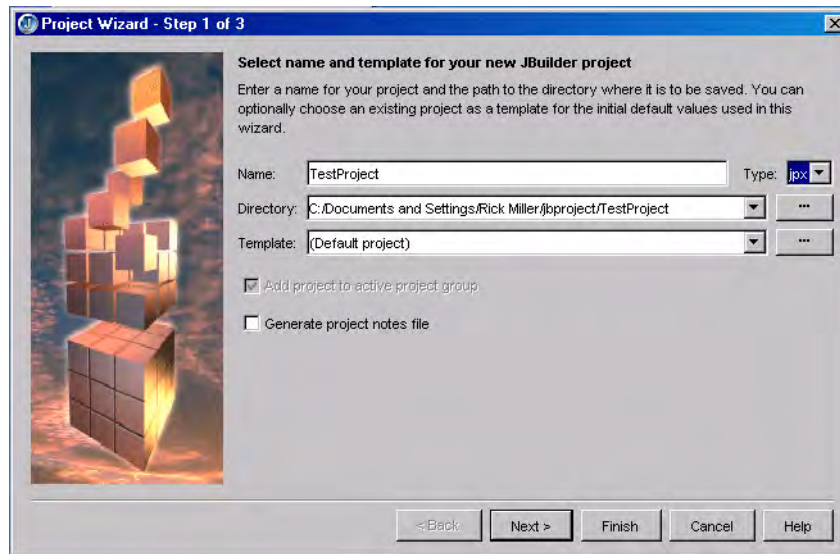


Figure 2-17: Project Wizard Step 1 of 3

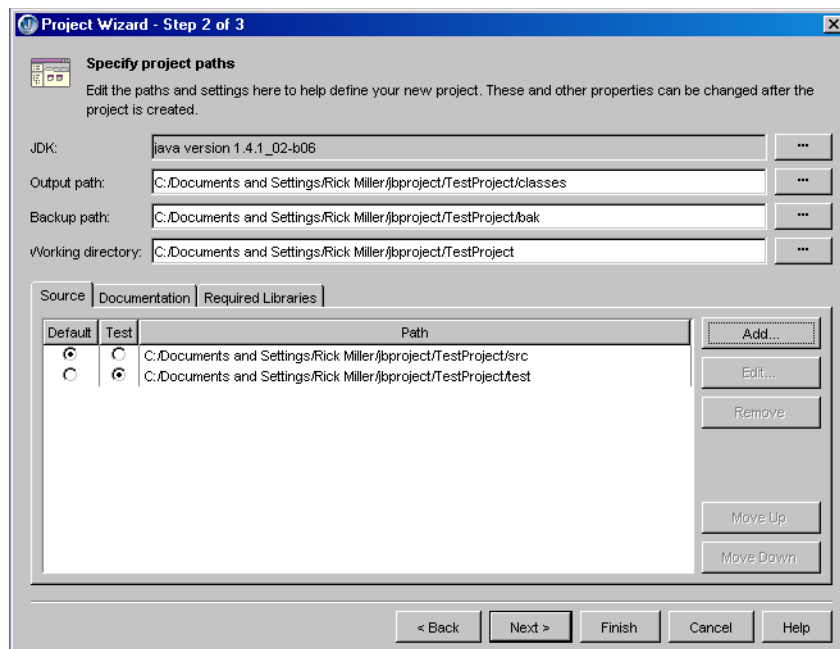


Figure 2-18: Project Wizard Step 2 of 3

STEP 2: CREATE OR Add SOURCE FILES, PACKAGES, OR OTHER COMPONENTS

Now that you have created the project, you are ready to add source files. This section shows you how to create a Java source file using the JBuilder Class Wizard. Select File > New Class. This will open the Class Wizard dialog window as shown in figure 2-20.

The Class Wizard lets you set the package name, class name, base class, and other options. For this example, I have entered *com.pulpfreepress.jfa.chapter1* as the package name, *ApplicationClass* as the class name, and used the JBuilder default recommendation *java.lang.Object* as the base class. (In Java, if you don't explicitly extend another class, you implicitly extend *java.lang.Object*.)

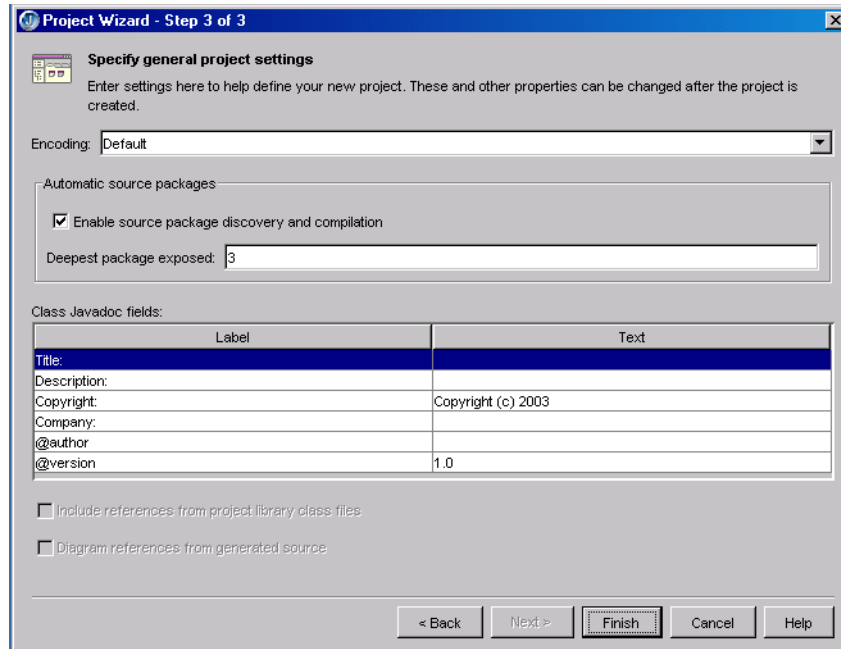


Figure 2-19: Project Wizard Step 3 of 3

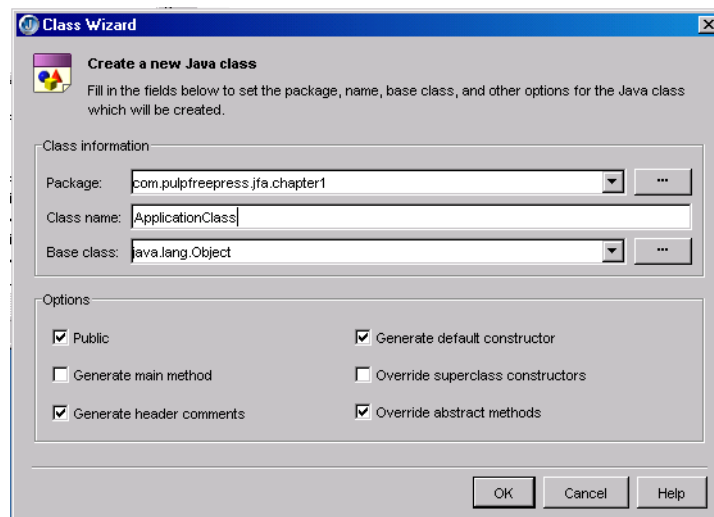


Figure 2-20: Creating ApplicationClass.java Source File

For the options, I've accepted the JBuilder recommend defaults. When you are satisfied with the Class Wizard settings, click the OK button to create the source file. At this point, you will have a new source file named ApplicationClass.java added to the TestProject in the com.pulpfreepress.jfa.chapter1 package. The ApplicationClass.java file will have a few lines of automatically generated source code. This will generally include the package declaration, class declaration, any default import statements, and some comments. You can modify this file so that it looks like the code given in example 2.2.

After you have created the ApplicationClass.java file, you must now create the SampleClass.java file in the same fashion. When you have finished creating both source files, your TestProject will look similar to that shown in figure 2-21.

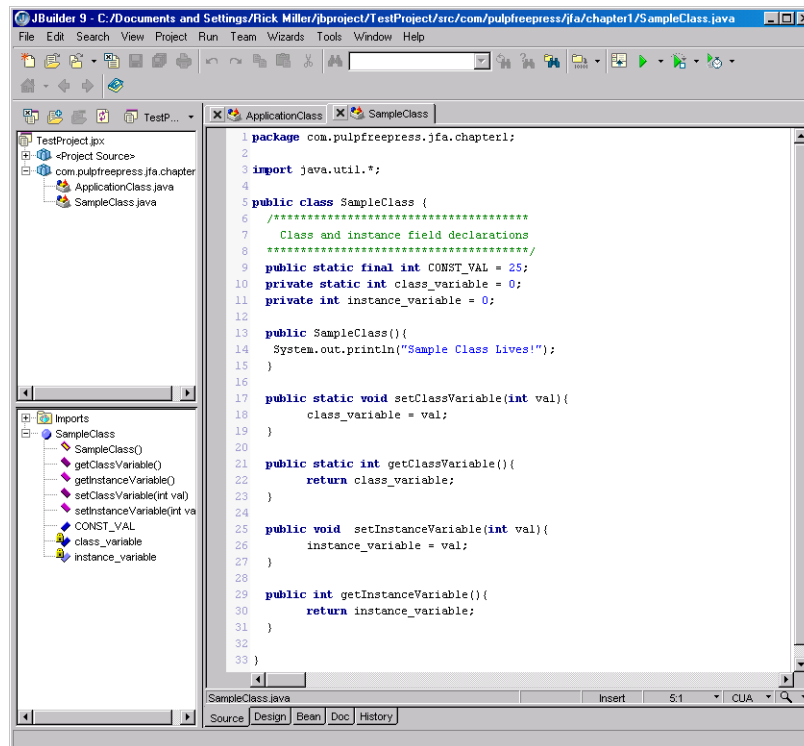


Figure 2-21: TestProject After Creating SampleClass.java and ApplicationClass.java

STEP 3: SET PROJECT PROPERTIES

Before running a JBuilder project, you may have to set some project properties. Some of these properties were automatically set when you created TestProject using the Project Wizard previously in step 1. These included project paths and javadoc comments. The number of properties that need to be configured will vary depending on the type of project and the version of JBuilder you are using.

To set or change a project's properties select Project > Project Properties. The Project Properties dialog window will open as is shown in figure 2-22. The property we are most concerned about setting in this example is the project's runtime configuration. This property is discussed in detail in the next step.

STEP 4: IDENTIFY THE APPLICATION CLASS

To run the TestProject, you must tell JBuilder which class contains the main() method. Do this by selecting Project > Project Properties and clicking the Run tab. (Refer again to figure 2-22.) Click the New or Edit button to display the Runtime Configuration dialog window as is shown in figure 2-23.

The Runtime Configuration dialog window allows you to name the configuration setting, set the runtime type, and specify JVM and application parameters. You can also make debug settings and configure the JBuilder optimizer if you have that option installed.

For this example, all you need to set is the name of the class that contains the main() method, and that would be ApplicationClass. When you have finished making the runtime configuration setting, click the OK button. You are now ready to run the program. Well, almost!

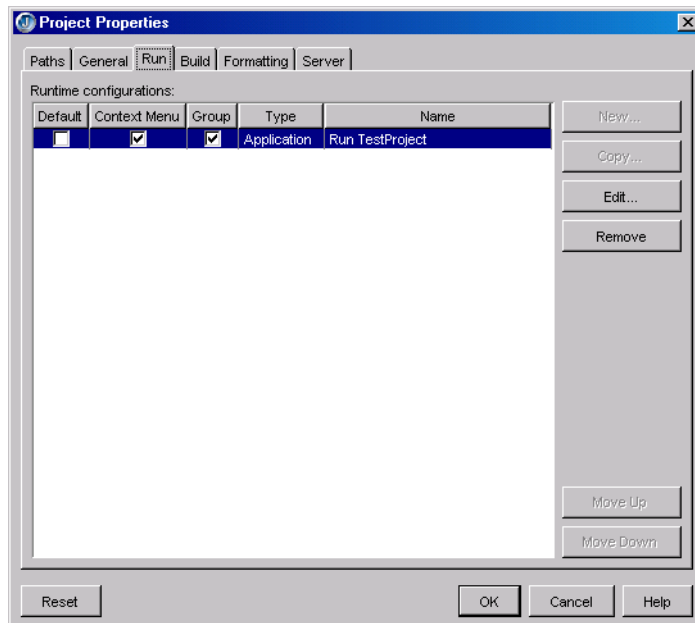


Figure 2-22: Project Properties Dialog

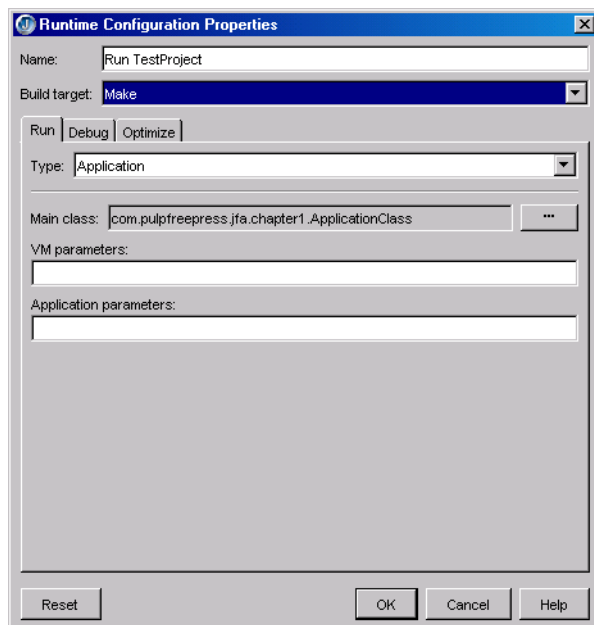


Figure 2-23: Runtime Configuration Properties Dialog

STEP 5: "MAKE" THE PROJECT

You are almost ready to run TestProject, but before you can do so, you must compile the source files. You can do this by “making” the project. This is where IDEs really shine. They handle all the messy details about project source files and their relationships to other files. IDEs like JBuilder track changes made to source files and will recompile any source files that have changed since the last project make.

Since this is the first attempt at running TestProject, you must first compile all its associated source files. You can do this either by selecting Project > Make Project or by pressing Ctrl+F9, as is shown in figure 2-24.

As the project builds, you will see one or more Build Progress dialog windows open on the screen. If all goes well, you will be ready to run the project once JBuilder completes the make process. However, don't be surprised if you have made a few mistakes when entering the source code. If there are problems with the project, JBuilder will report them to you. If this happens, you must edit the source files and correct the errors before attempting, once again, to make and run the project.

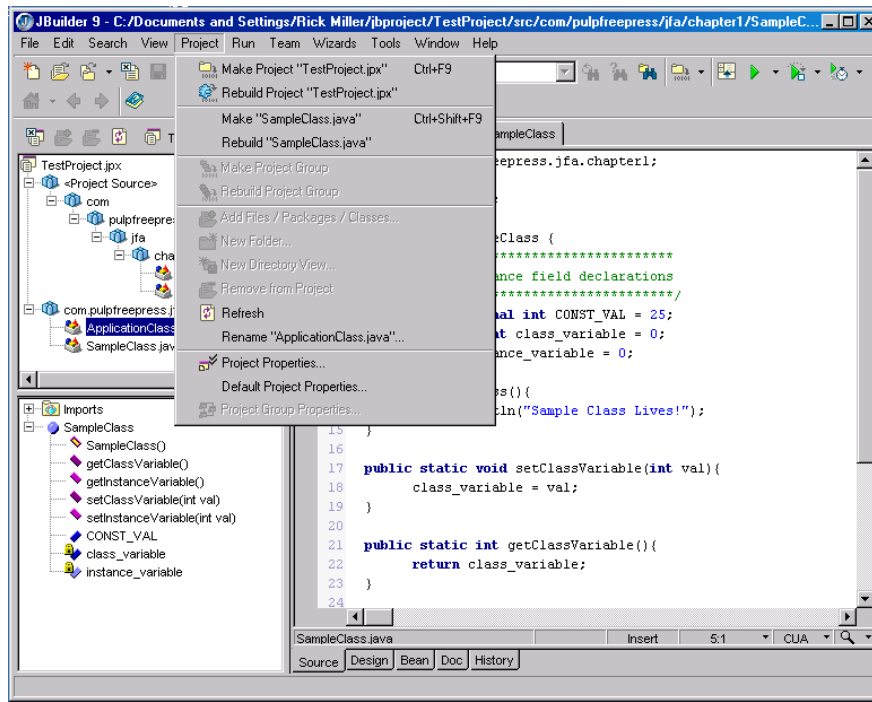


Figure 2-24: JBuilder Project Menu Showing the Make Project Item

STEP 6: RUN THE PROJECT

When TestProject builds successfully, you can run it either by selecting Run > Run Project or by pressing the F9 key. The results of running TestProject are shown in figure 2-25.

Quick Review

A Java IDE combines many software-development tools into one convenient package. Having a set of tightly-integrated development tools speeds software development. JBuilder is an excellent example of a Java IDE. JBuilder is project-centric. This means you begin the program-creation process with JBuilder by creating a project. JBuilder will then manage just about every aspect of the project from source-file creation to project execution.

CREATING JAVA PROJECTS USING LINUX™

The issues related to Java programming in the Linux environment are similar to those encountered with the Microsoft Windows operating system. You will need to obtain the Java SDK, install it, and configure your operating system's PATH and CLASSPATH environment variables. You will also need to select a suitable text editor or IDE. Once you have properly installed the Java SDK on your Linux system you can create, compile, and run your Java programs just like you would in the Windows environment.

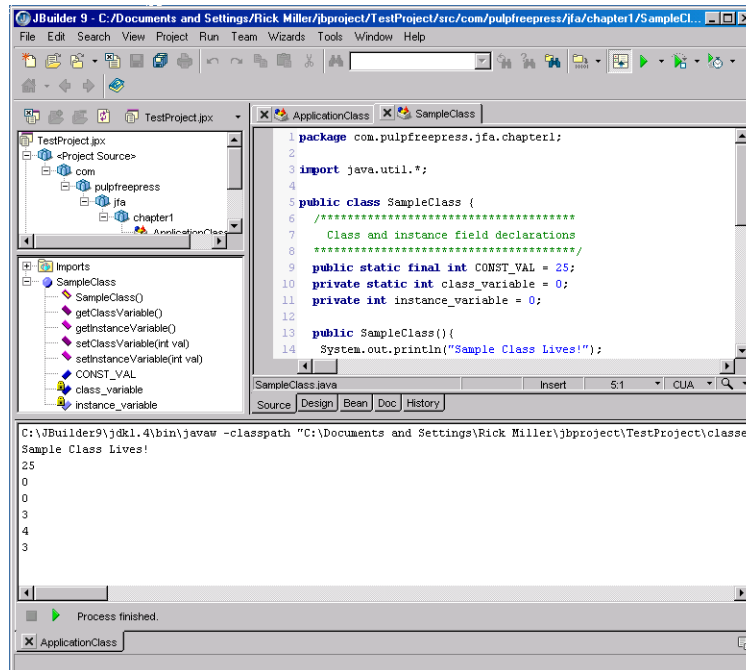


Figure 2-25: Running TestProject

This section assumes you are running Linux on a personally-owned computer for which you have root access. If you are using Linux at a school computer lab and need access to the Java programming tools, you must seek the help of the computer lab administrator.

A BRIEF NOTE ON THE MANY FLAVORS OF LINUX

Linux is a popular, open-source, UNIX-like operating system that comes in many flavors and targets several different processors. The examples used in this section are derived from RedHat™ Linux version 6.x running on a PC using an Intel iX86 or compatible processor.

As long as you run Linux on a PC, you should be OK when it comes to Java development. If your particular Linux distribution didn't come with Java developer tools, you can download the Java SDK from Sun. If, however, you are running Linux on a non-PC compatible computer, you may find that Java is not supported.

OBTAINING AND INSTALLING SUN'S J2SDK FOR LINUX

Most Linux distributions, such as RedHat, come with a set of developer tools that let you do software development on your particular system. For instance, RedHat ships with the GNU compiler tools g++, gcc, etc. For more information about the GNU project, visit the gnu.org website [<http://www.gnu.org/gnu/thegnuproject.html>].

If your distribution fails to include Java, or if you want to upgrade your Linux Java version, you can obtain the SDK from Sun's Java website [www.java.sun.com].

When you open the Java download page, you'll see two SDK download options: a self-extracting file and a self-extracting RPM module. For this example, I downloaded the self-extracting file. If you choose to download the self-extracting RPM module, you will have to consult the RPM utility documentation for installation instructions.

Figure 2-26 shows the j2sdk-1_4_2_02-linux-i586.bin file being saved to the /usr/local directory.

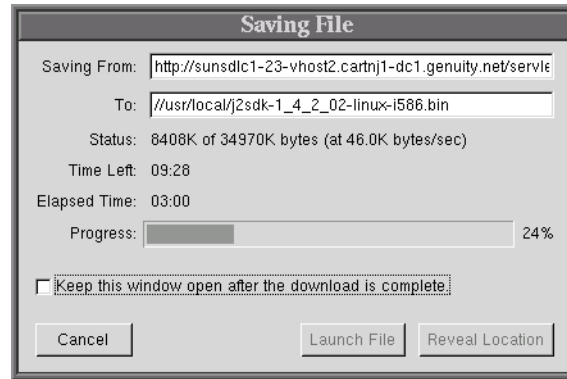


Figure 2-26: Downloading a Linux Java Self-Extracting Bin File

SELECTING AN INSTALLATION LOCATION

After you download the j2sdk self-extracting bin file, you will have to decide on an installation directory. I recommend creating a Java subdirectory in the /usr/local directory. To create the directory, move to the /usr/local directory and type the following command:

```
mkdir Java
```

After you create the Java directory, move or copy the j2sdk self-extracting bin file to the Java directory to perform the extraction. To extract the SDK in the Java directory, type the following command:

```
sh *.bin
```

Figure 2-27 shows the self-extracting file being executed. When execution begins, you are presented with a license agreement. After reviewing and agreeing to the license agreement, the Java package-extraction process automatically begins. When the self-extracting file completes execution, you will see a new subdirectory as is shown in figure 2-28.



Figure 2-27: Executing the Self-Extracting Bin File

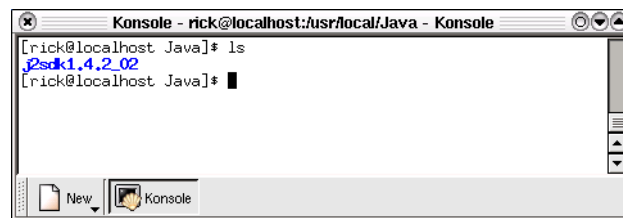
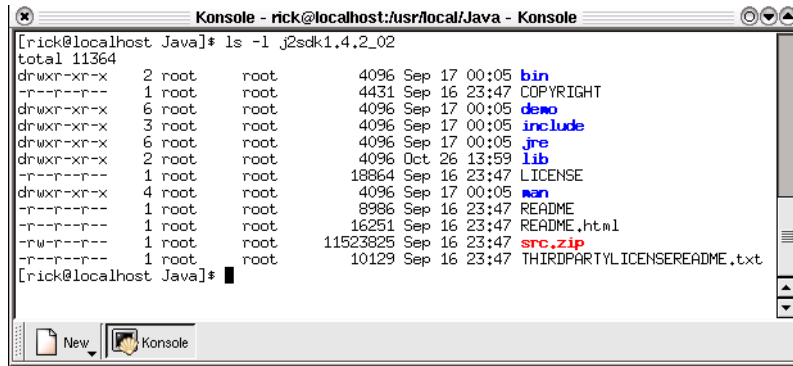


Figure 2-28: New Subdirectory Resulting from SDK Extraction

Figure 2-29 shows the contents of the j2sdk1.4.2_02 directory. Make a note of the path to the bin directory. If you installed the SDK in the /usr/local/Java directory the full path to the bin directory will be:

```
/usr/local/Java/j2sdk1.4.2_02/bin
```

You will need to know this path to set the PATH environment variable



```

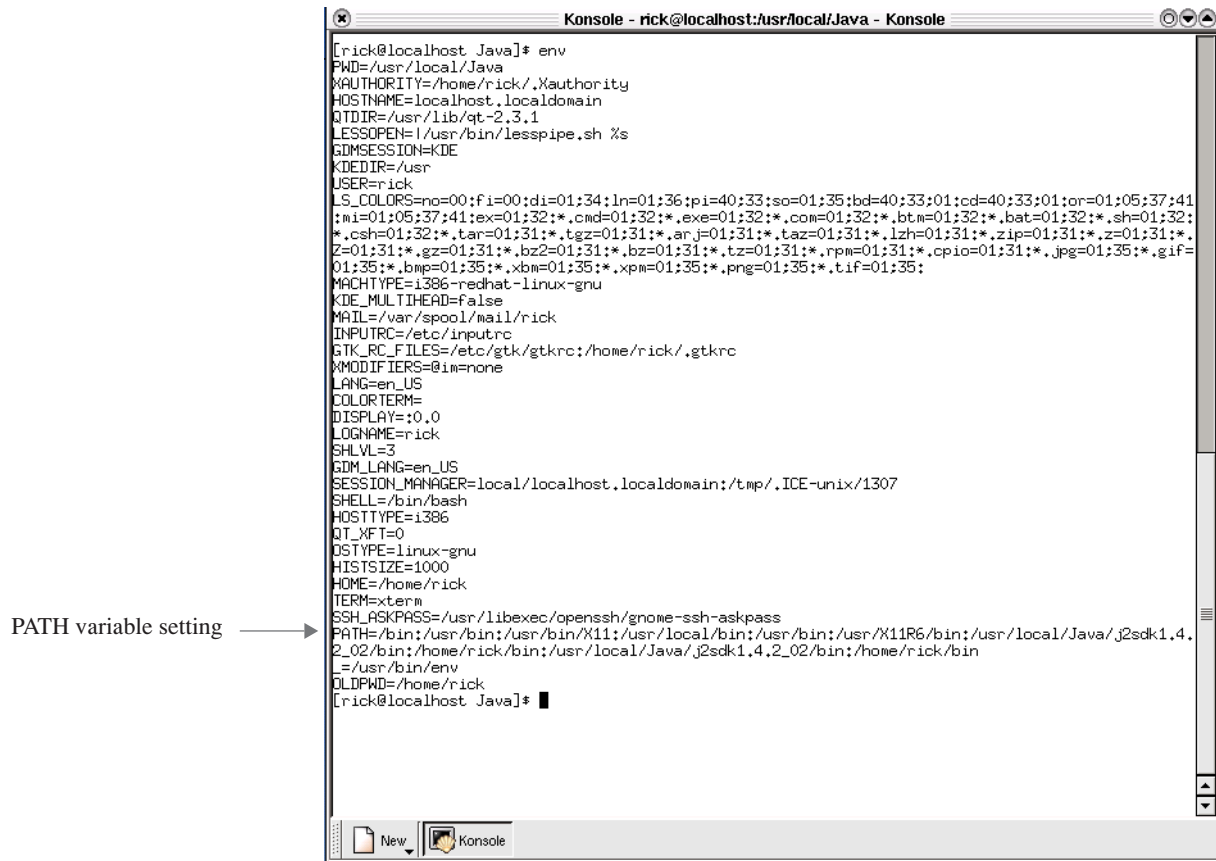
[rick@localhost Java]$ ls -l j2sdk1.4.2_02
total 11364
drwxr-xr-x  2 root  root    4096 Sep 17  00:05 bin
-r--r--r--  1 root  root    4431 Sep 16  23:47 COPYRIGHT
drwxr-xr-x  6 root  root    4096 Sep 17  00:05 demo
drwxr-xr-x  3 root  root    4096 Sep 17  00:05 include
drwxr-xr-x  6 root  root    4096 Sep 17  00:05 jre
drwxr-xr-x  2 root  root    4096 Oct 26  13:59 lib
-r--r--r--  1 root  root   18864 Sep 16  23:47 LICENSE
drwxr-xr-x  4 root  root    4096 Sep 17  00:05 man
-r--r--r--  1 root  root    8986 Sep 16  23:47 README
-r--r--r--  1 root  root   16251 Sep 16  23:47 README.html
-rw-r--r--  1 root  root  11523825 Sep 16  23:47 src.zip
-r--r--r--  1 root  root    10129 Sep 16  23:47 THIRDPARTYLICENSEREADME.txt
[rick@localhost Java]$

```

Figure 2-29: Contents of the j2sdk1.4.2_02 Directory

SETTING THE PATH ENVIRONMENT VARIABLE (BASH SHELL)

After installing the Java SDK, you need to set your shell's PATH environment variable. The PATH environment variable is used by the shell to locate executable commands. The PATH environment variable is already set to some default value. Most likely it is set to include the /bin and the /usr/bin directories. To check the PATH variable's current setting, use the `env` command as is shown in figure 2-30.



```

[rick@localhost Java]$ env
PWD=/usr/local/Java
XAUTHORITY=/home/rick/.Xauthority
HOSTNAME=localhost.localdomain
QTDIR=/usr/lib/qt-2.3.1
LESSOPEN=/usr/bin/lesspipe.sh %s
GDMSESSION=KDE
KDEDIR=/usr
USER=rick
LS_COLORS=no=00:fi=00:di=01:34:ln=01:36:pi=40:33:so=01:35:bd=40:33:01:cd=40:33:01:or=01:05:37:41
:mi=01:05:37:41:ex=01:32:*.cmd=01:32:*.exe=01:32:*.com=01:32:*.bat=01:32:*.sh=01:32:
*.csh=01:32:*.tar=01:31:*.tgz=01:31:*.arj=01:31:*.taz=01:31:*.lzh=01:31:*.zip=01:31:*.z=01:31:*.
Z=01:31:*.gz=01:31:*.bz2=01:31:*.bz=01:31:*.tz=01:31:*.rpm=01:31:*.cpio=01:31:*.jpg=01:35:*.gif=
01:35:*.bmp=01:35:*.xbm=01:35:*.xpm=01:35:*.png=01:35:*.tif=01:35:
MACHINE_TYPE=i386-redhat-linux-gnu
KDE_MULTIHEAD=false
MAIL=/var/spool/mail/rick
INPUTRC=/etc/inputrc
GTK_RC_FILES=/etc/gtk/gtkrc:/home/rick/.gtkrc
XMODIFIERS=@im=none
LANG=en_US
COLORTERM=
DISPLAY=:0.0
LOGNAME=rick
SHLVL=3
GDM_LANG=en_US
SESSION_MANAGER=local/localhost.localdomain:/tmp/.ICE-unix/1307
SHELL=/bin/bash
HOSTTYPE=i386
QT_XFT=0
OSTYPE=linux-gnu
HISTSIZE=1000
HOME=/home/rick
TERM=xterm
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
PATH=/bin:/usr/bin:/usr/bin/X11:/usr/local/bin:/usr/bin:/usr/X11R6/bin:/usr/local/Java/j2sdk1.4.
2_02/bin:/home/rick/bin:/usr/local/Java/j2sdk1.4.2_02/bin:/home/rick/bin
_=/usr/bin/env
OLDPWD=/home/rick
[rick@localhost Java]$

```

PATH variable setting →

Figure 2-30: Checking Environment Variables with the env Command

As you can see from figure 2-30, the *env* command displays a lot of information. Inspect your listing for the *PATH* environment variable setting.

To set the *PATH* environment variable, you will need to *su* as root and edit the bash *profile* file located in the */etc* directory. Add the following line to the end of the file:

```
export PATH=$PATH:/usr/local/Java/j2sdk1.4.2_02/bin
```

When you've finish editing the profile file, save the changes and restart Linux. To test the *PATH* variable, use the *env* command again and inspect the *PATH* variable as shown in figure 2-30, or type *java -version* at the prompt.

SETTING THE CLASSPATH ENVIRONMENT VARIABLE (BASH)

You can set the *CLASSPATH* environment variable the same way you set the *PATH* variable, by editing the bash *profile* file. The *CLASSPATH* variable can be set by adding the following command to the end of the profile file:

```
export CLASSPATH=$CLASSPATH:/usr/local/Java/j2sdk1.4.2_02/jre/lib:./:/classes
```

CREATING, COMPILING, AND RUNNING JAVA PROGRAMS

With the SDK installed, and the *PATH* and *CLASSPATH* environment variables set, you are ready to create, compile, and run Java programs. To do this, you will need to create the Java source files using a text editor, compile the resulting source files with the *javac* command, and then execute the main application using the *java* command. I discuss these steps in detail in the following sections.

CREATING JAVA SOURCE FILES

To create Java source files you'll need a suitable text editor. RedHad® Linux ships with several text editors that can do the job: *vi* and *emacs*.

vi

Vi is a compact yet powerful text editor used by professional programmers all over the world. If you are a novice Linux user you may find *vi* to be somewhat cryptic upon startup, and its array of commands overwhelming. But, with a little practice, and the memorization of a few commands, you can quickly learn enough about *vi* to create and edit Java source files.

The important thing to know about *vi* is that it has two primary modes of operation: *command* and *input*. When you use *vi*, you need to keep in mind which mode you are in. When you enter *vi* commands, you need to be in the command mode; when you want to edit text, you need to be in the input mode.

EMACS

If you prefer to use the mouse and execute menu commands, then you may find *emacs* more user-friendly than *vi*. *Emacs* is an extremely powerful editing tool that can do more than just edit text. *Emacs* can be extended and customized. Unlike *vi*, *emacs* does not require the user to switch between command and input modes of operation. However, like *vi*, you will have to exert some effort to learn and understand how *emacs* works. But once you learn how to create, edit, and save files in *emacs*, you will be up and running in no time.

CREATING YOUR WORKING DIRECTORY

Before you create your Java source files, you should create a working directory for the Java project you are currently working on. This working directory would usually be created as a subdirectory in your home directory. For instance, when I log into my Linux system, my home directory is */home/rick*. I could create a working directory named "Projects", and in that directory create subdirectories named "Project1", "Project2", etc. How you set up your working directory environment is largely a matter of taste.

CREATE AND SAVE JAVA SOURCE FILES

Using the text editor of your choice, create and save the Java source files `SampleClass.java` and `ApplicationClass.java` given in examples 2.1 and 2.2 respectively. When you create these files, save them in your working directory using the required source-code package structure.

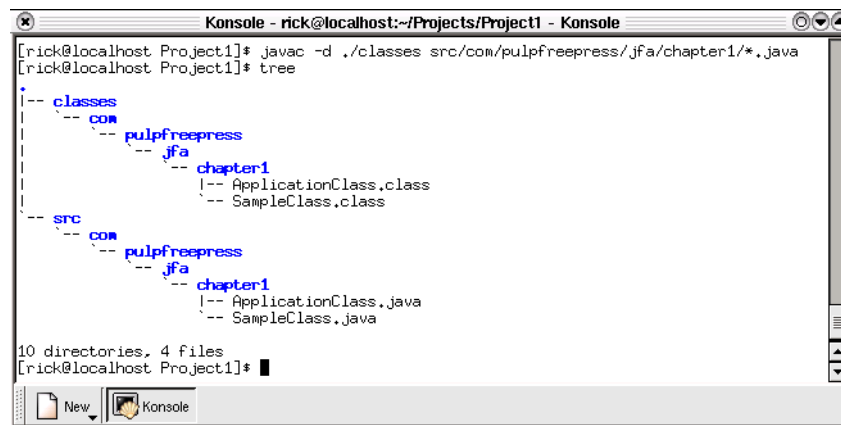
Compiling JAVA SOURCE FILES

After you have saved the source files you are ready to compile them. Use the `javac` compiler command to compile the `.java` files into `.class` files. You can compile the classes with the following command:

```
javac -d ./classes src/com/pulpfreepress/jfa/chapter1/*.java
```

If you have any compiler errors you will need to edit the source file again, fix the errors, and repeat the compilation step. When the source files successfully compile the resulting `.class` files will be automatically placed in their proper package structure in the `classes` subdirectory.

Figure 2-31 shows the `tree` command being used to reveal the directory structure of the `Project1` working directory after both `.java` source files have been compiled and the resulting `.class` files have been copied to the `com/pulpfreepress/jfa/chapter1` directory.



```
Konsole - rick@localhost:~/Projects/Project1 - Konsole
[rick@localhost Project1]$ javac -d ./classes src/com/pulpfreepress/jfa/chapter1/*.java
[rick@localhost Project1]$ tree
.
|-- classes
|   |-- com
|   |   |-- pulpfreepress
|   |   |   |-- jfa
|   |   |   |   |-- chapter1
|   |   |   |   |   |-- ApplicationClass.class
|   |   |   |   |   |-- SampleClass.class
|   |-- src
|   |   |-- com
|   |   |   |-- pulpfreepress
|   |   |   |   |-- jfa
|   |   |   |   |   |-- chapter1
|   |   |   |   |   |   |-- ApplicationClass.java
|   |   |   |   |   |   |-- SampleClass.java
|
+--- 10 directories, 4 files
[rick@localhost Project1]$
```

Figure 2-31: Using the `tree` Command to Show Directory Structure

RUNNING THE APPLICATION

You are now ready to run the program. From the `Project1` working directory type the following command to execute the main application class:

```
java com.pulpfreepress.jfa.chapter1.ApplicationClass
```

Figure 2-32 shows the results of running the `ApplicationClass` in the Linux environment.



```
Konsole - rick@localhost:~ - Konsole
[rick@localhost rick]$ java com.pulpfreepress.jfa.chapter1.ApplicationClass
Sample Class Lives!
25
0
0
3
4
3
[rick@localhost rick]$
```

Figure 2-32: Running `ApplicationClass` in the Linux Environment

Quick Review

The issues regarding Java development in the Linux environment are similar to those associated with the Windows operating system. You must obtain and install the Java SDK, configure the `PATH` and `CLASSPATH` environment variables, select a suitable text editor with which to create the Java source files, compile them using the `javac` compiler command, and run the main application with the `java` command.

The actual procedure for setting the `PATH` and `CLASSPATH` environment variables depends on your choice of Linux command shells. If you use a shell other than `sh` or `bash`, consult a reference for that shell.

CREATING JAVA PROJECTS USING MACINTOSH OS X DEVELOPER TOOLS

The Macintosh OS X operating system is based on an open-source version of Berkeley BSD UNIX called Darwin. So, if you are a Linux user, or you are familiar with another version of UNIX, you will feel right at home on a Macintosh running OS X. However, if you don't know a thing about UNIX, you will still feel right at home on a Macintosh because that's the beauty of a Macintosh!

OS X ships with a complete set of software-development tools, including the Java SDK and JRE. If you choose to do Java development on the Macintosh, you will get upgrades to the Java SDK directly from Apple Computer.

Once you install the OS X developer tools, you are ready to start Java development. The Java command-line tools are automatically installed in the `/usr/bin` directory. Your `PATH` and `CLASSPATH` environment variables will already be set to use the Java SDK.

You can develop Java programs using either the Java command-line tools or the OS X Xcode IDE. Xcode is a full-featured IDE that enables you to create programs not only in Java but in C, C++, and Objective C as well.

CREATING A JAVA PROJECT WITH XCODE

Xcode, like any IDE, is project-centric. Thus, the first step to creating a Java program in Xcode is to create a Java project. When you start Xcode, create a new project by selecting the `File > New Project` menu. This launches the Xcode New Project Assistant, as is shown in figure 2-33.

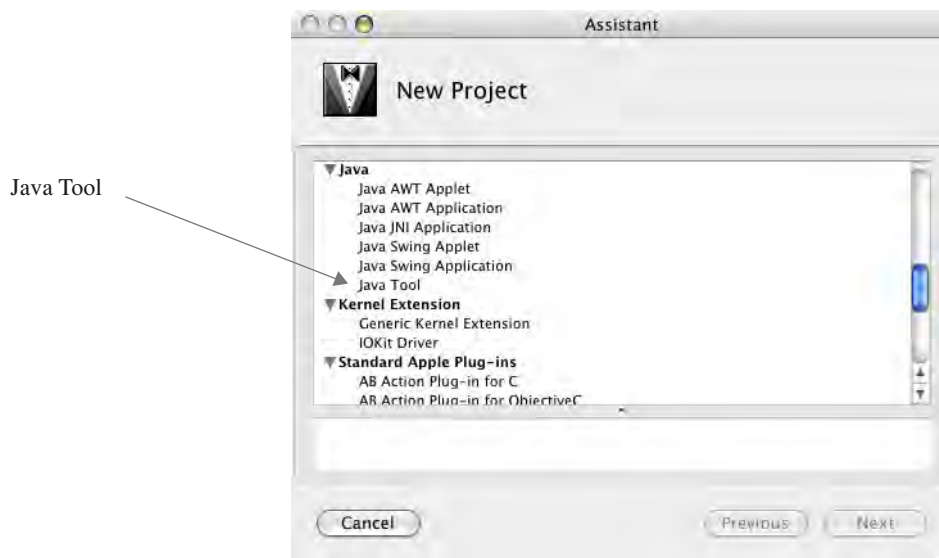


Figure 2-33: Xcode New Project Assistant

For this example select, Java Tool in the Assistant window, and click the Next button. This will take you to the New Java Tool window as is shown in figure 2-34.

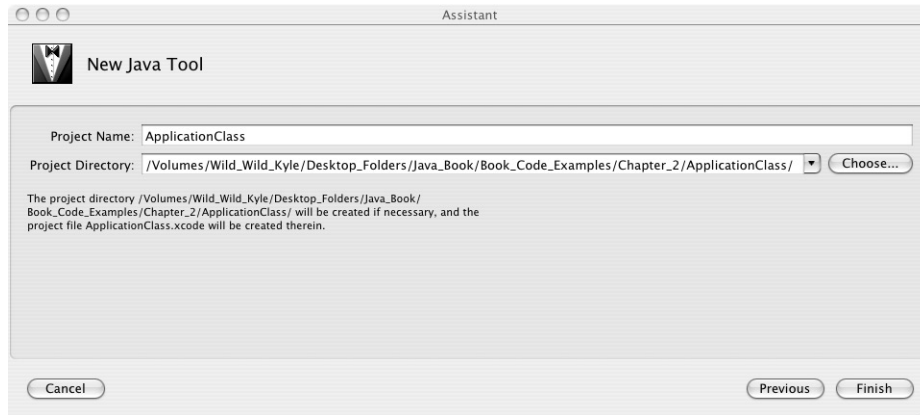


Figure 2-34: New Java Tool Window

Enter the name of the project in the Project Name field and use the Choose button to set or select the project directory. When you are finished with the New Java Tool window, click the Finish button. Xcode will automatically create both the project and a .java application source file bearing the project's name.

Edit the ApplicationClass.java file to make it look like example 2.2. When finished, you will need to add the SampleClass.java file. This is easy with Xcode: simply add a new Java class file to the project named SampleClass and edit it to look like example 2.1. Figure 2-35 shows the Xcode ApplicationClass project with both ApplicationClass.java and SampleClass.java files added.

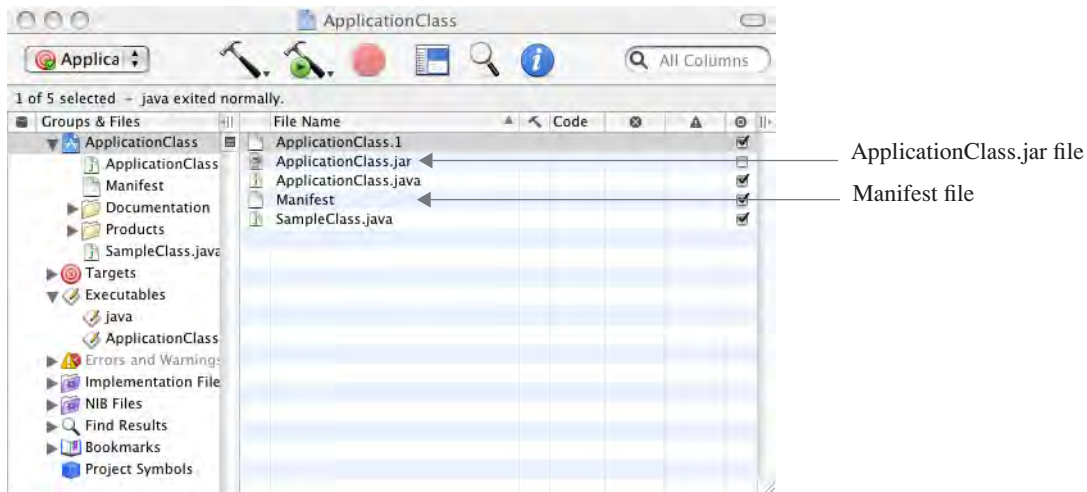


Figure 2-35: Xcode ApplicationClass Project Window

Notice that figure 2-35 shows the presence of two additional project artifacts. Xcode, by default, creates a .jar file when it builds the project. (A *jar file is a Java Archive File and is similar to a zip file.*) The manifest file is an important part of the .jar file, because it names the main application class. In this example, the manifest file needs to be edited to reflect the main application class's fully-qualified package name. Figure 2-36 shows the Manifest file being edited.

After editing the manifest file, build and run the project by selecting the Build > Build and Run menu command. Figure 2-37 shows the ApplicationClass project running in the Xcode environment.

Quick Review

Macintosh OS X provides a complete suite of Java programming tools. These range from traditional UNIX text editors such as vi and emacs, to the full-featured IDE Xcode. The Java SDK comes with OS X. Updates to the SDK must be obtained directly from Apple.



Figure 2-36: Editing the Manifest File to Reflect the Correct Package Location of the Main Application Class

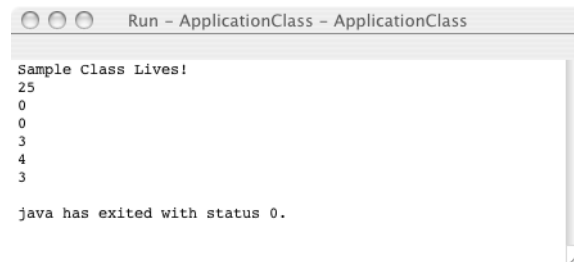


Figure 2-37: Running ApplicationClass Project from Xcode IDE

CREATING AND RUNNING EXECUTABLE JAR FILES

Java `.jar` files are archive files, like `.zip` files, that can contain a collection of Java artifacts including class files, source code, supporting documentation, etc. You can execute Java `.jar` files by adding a manifest to the `.jar` file that specifies the name of the main application class. This section provides step-by-step instructions that show you how to create and run an executable `.jar` file.

STEPS TO CREATE AN EXECUTABLE JAR FILE

The following steps are required to create an executable `.jar` file:

- **Step 1:** Create a Java program and save the resulting `.class` files in the proper package structure
- **Step 2:** Create a manifest file specifying the fully-qualified name of the main application class
- **Step 3:** Use the `jar` command to create the `.jar` file

STEP 1: CREATE YOUR JAVA PROGRAM

Using the Java development environment of your choice, create a Java project as described in the previous sections of this chapter. Be sure to save the resulting `.class` files in the proper package directory structure. For this example, I will use `SampleClass.class` and `ApplicationClass.class`. Both of these files are saved in the following package directory structure:

```
com/pulpfreepress/jfa/chapter1
```

The only issue to note here is that if you are using Microsoft Windows the slashes would be replaced with backslashes like this:

```
com\pulpfreepress\jfa\chapter1
```

STEP 2: CREATE MANIFEST FILE

Using a text editor create a text file that contains a line specifying the fully-qualified name of the main application class. In this example, the fully-qualified name of `ApplicationClass` is this:

```
com.pulpfreepress.jfa.chapter1.ApplicationClass
```

Therefore, the entry in the manifest file would then look like this:

```
Main-class: com.pulpfreepress.jfa.chapter1.ApplicationClass
```

(IMPORTANT NOTE: Type a carriage return after this line. Failure to do so results in some Microsoft Windows systems reporting a “Failed to load Main-Class manifest attribute” error.)

Save the manifest file using any filename you choose. You do not have to provide a filename extension. For this example, I saved the manifest file with the name “mainclass”.

STEP 3: USE THE JAR COMMAND

Now that you have created the manifest file, you are ready to use the `jar` command. The `jar` command takes the following format:

```
jar cmf [manifest filename] [jar filename] [package to jar]
```

c = create new archive
m = include manifest information from specified manifest file
f = name of the jar file to create

According to this `jar` command format, if you created the `com` package structure in your working directory, and you created the manifest file named “mainclass”, and you wanted to name the jar file “MyApp.jar”, the actual `jar` command would look like this:

```
jar cmf mainclass MyApp.jar com
```

This would use the `mainclass` manifest file to create the `MyApp.jar` file using the class files located in the `com.pulpfreepress.jfa.chapter1` package.

EXECUTING THE JAR FILE

To execute the `.jar` file, use the `java` command with the `-jar` switch as is shown here:

```
java -jar MyApp.jar
```

Figure 2-38 shows the `MyApp.jar` file being executed with the `java` command and the `-jar` option.

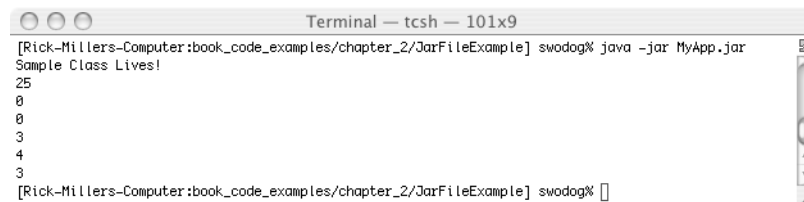


Figure 2-38: Executing the `MyApp.jar` File Using the `java` Command

Quick Review

The Java `jar` command lets you create executable `.jar` files. To create an executable `.jar` file you need to create a manifest file that provides the fully-qualified name of the main application class. Be sure to add an extra line to the manifest so that it work in the Microsoft Windows environment. To execute the `.jar` file use the `java` command with the `-jar` option.

SUMMARY

A Java IDE combines many software-development tools into one convenient package. Using a set of tightly-integrated development tools speeds software development. JBuilder is an excellent example of a Java IDE. JBuilder is project-centric. This means you begin the program-creation process with JBuilder by creating a project. JBuilder will then manage just about every aspect of the project, from source-file creation to project execution.

The issues regarding Java development in the Linux environment are similar to those associated with the Microsoft Windows operating system. You must obtain and install the Java SDK, configure the PATH and CLASSPATH environment variables, select a suitable text editor with which to create the Java source files, compile them using the *javac* compiler command, and run the main application with the *java* command.

The actual procedure for setting the PATH and CLASSPATH environment variables depends on your choice of Linux command shells. If you use a shell other than *sh* or *bash*, consult a reference for that shell.

Macintosh OS X provides a complete suite of Java programming tools. These range from traditional UNIX text editors such as *vi* and *emacs* to the full-featured IDE Xcode. The Java SDK comes with OS X. Updates to the SDK must be obtained directly from Apple.

You can create executable *.jar* files with the *jar* command-line tool. To create an executable *.jar* file you need to create a manifest file that provides the fully-qualified name of the main application class. Be sure to add an extra line to the manifest so that it work in the Microsoft Windows environment. Execute the *.jar* file using the *java* command with the *-jar* option.

Skill-Building Exercises

- 1. Obtain and Install the J2SDK:** Obtain the Java 2 Standard Edition Software Development Kit from Sun and install on your computer. Note the name of the SDK install directory for future reference.
- 2. Operating System Configuration:** Ensure your operating system PATH and CLASSPATH environment variables are set properly. Refer to the sections in this chapter that relate to your operating system for instructions on how to set them.
- 3. Test the SDK:** After you have installed the SDK and have configured your operating system's PATH and CLASSPATH environment variables, test the SDK by typing *javac* at a command or terminal window. If you did not get the expected results, re-check your environment variable settings and keep at it until you get it working.
- 4. Obtain and Install a Programmer's Text Editor:** Research the various programmer's text editors available for your computer, select one, and install it. Some features to look for include syntax highlighting and Java command-line tool integration.
- 5. Obtain and Install an Integrated Development Environment:** Research the various IDEs available for your computer, and obtain one that's appropriate for your development situation. Your decision may be driven by various factors such as school or work requirements or personal preference.
- 6. Create, Compile, and Run a Java Project:** After you have installed and configured your development environment, create a Java project using the source files given in examples 2.1 and 2.2. The objective of this exercise is to get you comfortable with using your chosen development environment.

SUGGESTED PROJECTS

1. **Operating System Commands:** During your Java programming career you will often be required to interact directly with your computer's operating system via a set of operating-system commands. Obtain an operating system command reference book for your operating system (*i.e.*, *Unix/Linux Terminal or Windows Command Prompt*) and study the commands. The following table lists several helpful MS-DOS and UNIX commands.

MS-DOS	UNIX	Meaning
	man, xman	Manual commands to display UNIX system documentation on the terminal.
	info	Displays documentation on Linux utilities
help help <command> <command> /?		Displays a list of MS-DOS commands and their use Gets help for a specific command Gets help for a specific command
dir	ls	List directory contents
copy	cp	copy file
rename	mv	change the name of a file
erase, del	rm	remove file
env	set	get information about system environment setup

Table 2-1: Helpful Operating System Commands

2. **Java Command-Line Tools:** Familiarize yourself with all the Java command-line tools that come with the Java SDK. These tools are found in the <JAVA_HOME>\bin directory where JAVA_HOME is an environment variable that represents the path to the J2SDK installation directory.
3. **Gather Java Documentation:** Obtain a good Java quick reference guide. My favorite is the *Java In A Nutshell* series by O'Reilly, and is listed in the references section at the end of this chapter. You can also access the latest Java SDK and API documentation directly from the [java.sun.com] website.

SELF-TEST QUESTIONS

1. What is the purpose of the PATH environment variable in the Windows and UNIX operating system environments?
2. What is the purpose of the CLASSPATH environment variable in the Windows and UNIX operating system environments?
3. List and describe the steps of the Java program-creation process.
4. What are the primary components of an IDE? What benefits would the use of an IDE offer a programmer?
5. What is the purpose and use of the *javac* command?
6. What is the purpose and use of the *java* command?

7. What is the purpose and use of the *javadoc* command?
8. List and describe the steps required to create an executable .jar file.
9. True/False: The *javac* command can compile more than one source file at a time.
10. How do you execute an executable .jar file?

REFERENCES

Mark G. Sobell. *A Practical Guide to Linux*. Addison-Wesley, Reading, Massachusetts, 1997. ISBN: 0-201-89549-8

Matisse Enzer. *UNIX For Mac OS X*. Peachpit Press, Berkeley, California, 2003, ISBN: 0-201-79535-3

Java Online Documentation for the J2SDK. [java.sun.com]

David Flanagan. *JAVA In A Nutshell: A Desktop Quick Reference*, 4th Edition. O'Reilly & Associates, Inc., Sebastopol, California, 2002. ISBN: 0-596-00283-1

NOTES

CHAPTER 3



BEACH TREES

PROJECT WALKTHROUGH: A COMPLETE EXAMPLE

LEARNING OBJECTIVES

- *Apply the project-approach strategy to implement a Java programming assignment*
- *Apply the development cycle to implement a Java programming assignment*
- *State the actions performed by the development roles analyst, designer, and programmer*
- *Translate a project specification into a software design that can be implemented in Java*
- *State the purpose and use of method stubbing*
- *State the purpose and use of state transition diagrams*
- *Explain the concept of data abstraction and the role it plays in the design of user-defined data types*
- *Use the javac command-line tool to compile Java source files*
- *Execute Java programs using the java command-line tool*
- *State the importance of compiling and testing early in the development process*
- *Use the javadoc command-line tool to create professional-grade documentation*

INTRODUCTION

This chapter presents a complete example of the analysis, design, and implementation of a typical classroom programming project. The objective of this chapter is to demonstrate to you how to approach a project and, with the help of the project-approach strategy and the development cycle, formulate and execute a successful project implementation plan.

The approach I take to the problem solution is procedural based. I do this because I find that trying simultaneously to learn problem-solving skills, the Java command-line tools, how to employ the development cycle, and object-oriented design and programming concepts, is simply too overwhelming for most students to bear. However, try as I may to defer the discussion of object-oriented concepts, Java forces the issue by requiring that all methods belong to a class. I mitigate this by pursuing a solution that results in one user-defined class. As you pursue your studies of Java and progress through the remaining chapters of this book, you will quickly realize that there are many possible solutions to this particular programming project, some of which require advanced knowledge of object-oriented programming theory and techniques.

You may not be familiar with some of the concepts discussed here. Don't panic! I wrote this material with the intention that you revisit it when necessary. As you start writing your own projects examine these pages for clues on how to approach your particular problem. In time, you will begin to make sense of all these confusing concepts. Practice breeds confidence, and after a few small victories, you will never have to refer to this chapter again.

THE PROJECT-APPROACH STRATEGY SUMMARIZED

The project-approach strategy presented in chapter 1 is summarized in table 3-1. Keep the project-approach strategy in mind as you formulate your solution. Remember, the purpose of the project-approach strategy is to kick-start the creative process and sustain your creative momentum. Feel free to tailor the project-approach strategy to suit your needs.

Strategy Area	Explanation
Application Requirements	Determine and clarify exactly what purpose and features the finished project must have. Clarify your understanding of the requirements with your instructor if the project specification is not clear. <i>This results in a clear problem definition and a list of required project features.</i>
Problem Domain	Study the problem until you have a clear understanding of how to solve it. Optionally, express your understanding of the solution by writing a pseudocode algorithm that describes, step-by-step, how you will solve the problem. You may need to do this several times on large, complex projects. <i>This results in a high-level solution statement that can be translated into an application design.</i>
Language Features	Make a list of all the language features you must understand and use to draft a competent design and later implement your design. As you study each language feature check it off your list. Doing so will give you a sense of progress. <i>This results in a notional understanding of the language features required to effect a good design and solve the problem.</i>
High-Level Design & Implementation Strategy	Sketch out a rough application design. A design is simply a statement, expressed through words, pictures, or both, of how you plan to implement the problem solution derived in the Problem Domain strategy area. <i>This results in a plan of attack!</i>

Table 3-1: Project Approach Strategy

DEVELOPMENT CYCLE

When you reach the project design phase you will begin to employ the *development cycle*. It is good to have a broad, macro-level design idea to get you started, but don't make the mistake of trying to design everything up front. Design until you can begin coding and test some of your design ideas. The development cycle is summarized in table 3-2.

Step	Explanation
Plan	Design to the point where you can get started on the implementation. Do not attempt to design everything up front. The idea here is to keep your design flexible and open to change.
Code	Implement what you have designed.
Test	Thoroughly test each section or module of source code. The idea here is to try to break it before it has a chance to break your application. Even in small projects you will find yourself writing short test-case programs on the side to test something you have just finished coding.
Integrate/Test	Add the tested piece of the application to the rest of the project and then test the whole project to ensure it didn't break existing functionality.
Refactor	This step applies more to object-oriented programming than to procedural programming. It means to take a comprehensive look at your overall application architecture and migrate general functionality up into base, or even abstract, classes so the functionality can be utilized by more concrete derived classes.

Table 3-2: Development Cycle

Employ the development cycle in a tight spiral fashion as depicted in figure 3-1. By tight spiral I mean you will begin with the plan step, followed by the code step, followed by the test step, followed by the integrate step, optionally followed by the refactor step. When you have finished a little piece of the project in this fashion, you return to the plan step and repeat the process. Each complete plan, code, test, integrate, and refactor sequence is referred to as an *iteration*. As you iterate through the cycle you will begin to notice the time it takes to complete the cycle from the beginning of the plan step to the completion of the integrate step decreases. The development cycle spirals tighter and tighter as development progresses until you converge on the final solution.

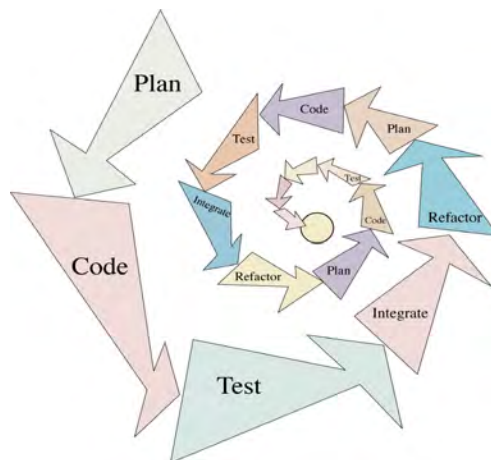


Figure 3-1: Tight Spiral Development Cycle Deployment

PROJECT SPECIFICATION

Keeping both the project-approach strategy and development cycle in mind, let's look now at a typical project specification given in table 3-3.

IST 149 - Introduction To Java Programming Project 1 Robot Rat
<p>Objectives:</p> <p>Demonstrate your ability to utilize the following language features:</p> <ul style="list-style-type: none"> Primitive, reference, and array data types Two-dimensional arrays Program flow-control structures Class methods Class attributes Local method variables Constructor methods Console input and output Java applications <p>Task:</p> <p>You are in command of a robot rat! Write a Java application that will allow you to control the rat's movements around a 20 x 20 grid floor.</p> <p>The robot rat is equipped with a pen. The pen has two possible positions, up or down. When in the up position, the robot rat can move about the floor without leaving a mark.</p> <p>If the pen is down the robot rat leaves a mark as it moves through each grid position. Moving the robot rat about the floor with the pen up or down at various locations will result in a pattern written upon the floor.</p> <p>Hints:</p> <ul style="list-style-type: none"> - The robot rat can move in four directions: north, south, east, and west. - Implement the floor as a two-dimensional array of boolean. - Java provides the <code>System.out.println()</code> method that makes it easy to write text to the console, but getting text input from the console is not as straightforward. Look to the <code>java.io</code> package's <code>InputStreamReader</code> and <code>BufferedReader</code> classes for help. <p>At minimum, provide a text-based command menu with the following or similar command choices:</p> <ol style="list-style-type: none"> 1. Pen Up 2. Pen Down 3. Turn Right 4. Turn Left 5. Move Forward 6. Print Floor 7. Exit

Table 3-3: Project Specification

Another valid requirements question might focus on exactly what is meant by the term *Java application*. That too is a good question. A Java application is a class that contains a `main()` method. You could write this program as a Java applet as well, although doing so is not a requirement for this project.

What about error checking? Again, good question. In the real world, making sure an application behaves well under extreme user conditions, and recovers gracefully in the event of some catastrophe, consumes the majority of the programming effort. One area in particular that requires extra measures to ensure everything goes well is array processing. As the robot rat is moved around the floor care must be taken to prevent the program from letting it go beyond the bounds of the floor array.

Something else to consider is how to process menu commands. Since the project only calls for simple console input and output, I recommend treating everything as a text string on input. If you need to convert a text string into another data type you can use the helper classes provided by the `java.lang` package. Otherwise, I want you to concentrate on learning how to use the fundamental language features listed in the project's objectives section, so I promise not to try to break your program when I run it.

You may safely assume, for the purposes of this project, that the user is perfect, yet noting for the record that this is absolutely not the case in the real world!

Summarizing the requirements thus far:

- You must write a program that models the concept of a robot rat and its movement upon a floor
- The robot rat is an abstraction represented by a collection of attributes, (*I discuss these attributes in greater detail in the problem domain section below.*)
- The floor is represented in the program as a two dimensional array of boolean
- Use just enough error checking, focusing on staying within the array boundaries
- Assume the user is perfect
- Read user command input as a text string
- Put all program functionality into one user-defined class. This class will be a Java application because it will contain a `main()` method

When you are sure you fully understand the project specification you can proceed to the problem domain strategy area.

PROBLEM DOMAIN STRATEGY AREA

In this strategy area your objective is to learn as much as possible about what a robot rat is and how it works in order to gain insight into how to proceed with the project design. A good technique to use to help jump-start your creativity is to go through the project specification and look for relevant nouns and verbs or verb phrases. A first pass at this activity will yield two lists. The list of nouns will suggest possible application objects, data types, and object attributes. Nouns will also suggest possible names for class and instance fields (*variables and constants*) and method variables. The list of verbs will suggest possible object interactions and method names.

NOUNS & VERBS

A first pass at reviewing the project specification yields the list of nouns and verbs shown in table 3-4.

Nouns	Verbs
robot rat	move
floor	set pen up
pen	set pen down
pen position (up, down)	mark
mark	turn right
program	turn left
pattern	print floor
direction (north, south, east, west)	exit
menu	

Table 3-4: Robot Rat Nouns and Verbs

This list of nouns and verbs is a good starting point, and now that you have it, what should you do with it? Good question. As I mentioned above, each noun is a possible candidate for either a variable, a constant, or some other data type, data structure, object, or object attribute within the application. A few of the nouns will not be used. Others will have a direct relationship to a particular application element. Some nouns will look like they could be very useful but may not easily convert or map to any application element. Also, the noun list may not be complete. You may discover additional application objects and object interactions as the project's analysis moves forward.

The verb list for this project example derives mostly from the suggested menu. Verbs normally map directly to potential method names. You will need to create these methods as you write your program. Each method you identify will belong to a particular object and may utilize some or all of the other objects, variables, constants, and data structures identified with the help of the noun list.

The noun list gleaned so far suggests that the robot rat project needs further analysis both to expand your understanding of the project's requirements and to reveal additional attribute candidates. How do you proceed? I recommend taking a closer look at several nouns that are currently on the list, starting with *robot rat*. Just what is a robot rat from the attribute perspective? Since pictures are always helpful I suggest drawing a few. Here's one for your consideration.

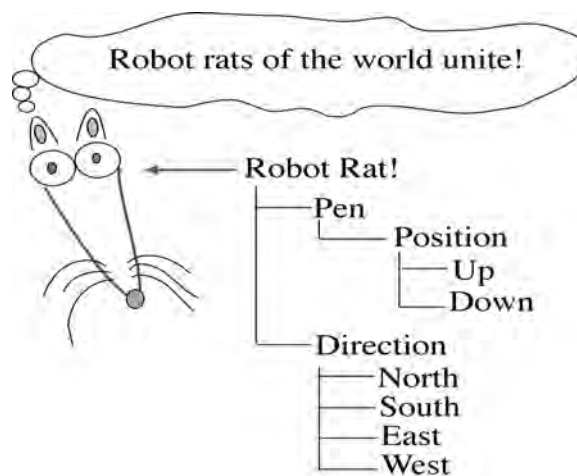


Figure 3-2: Robot Rat Viewed As Attributes

This picture suggests that a robot rat, as defined by the current noun list, consists of a pen that has two possible positions, and the rat's direction. As described in the project specification and illustrated in figure 3-2, the pen can be either *up* or *down*. Regarding the robot rat's direction, it can face one of four ways: *north*, *south*, *east*, or *west*. Can more attributes be derived? Perhaps another picture will yield more information. I recommend drawing a picture of the *floor* and run through several robot rat movement scenarios.

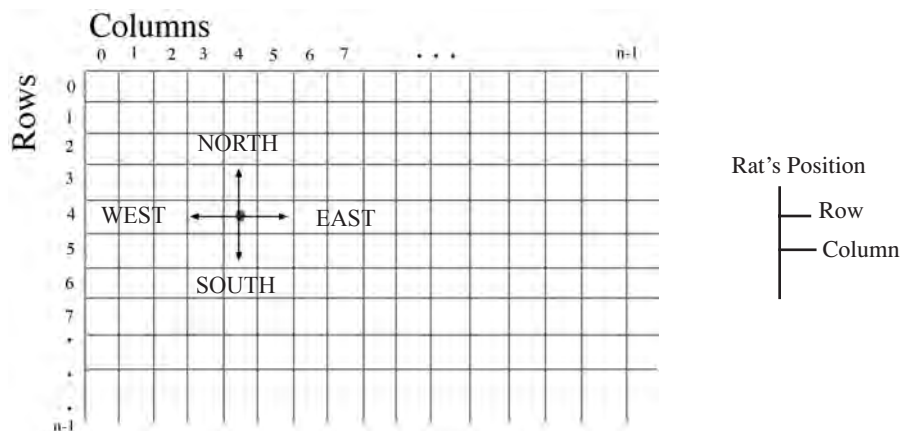


Figure 3-3: Robot Rat Floor Sketch

Figure 3-3 offers a lot of great information about the workings of a robot rat. The floor is represented by a collection of cells arranged by *rows* and *columns*. As the robot rat is moved about the floor its *position* can be determined by keeping track of its current *row* and *column*. These two nouns are good candidates to add to the list of relevant nouns and to the set of attributes that can be used to describe a robot rat. Before the robot rat can be moved, its current position on the floor must be determined, and upon completion of each move its current position must be updated. Armed with this information you should now have a better understanding of what attributes are required to represent a robot rat as figure 3-4 illustrates.

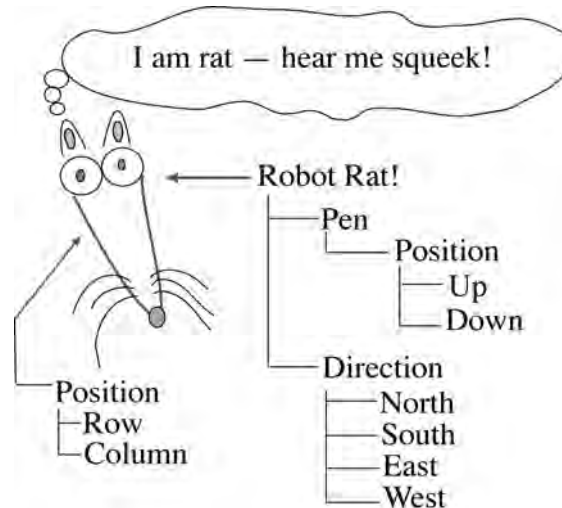


Figure 3-4: Complete Robot Rat Attributes

This seems to be a sufficient analysis of the problem at this point. You can return to this strategy area at any time should further analysis be required. It is now time to take a look at what language features must be understood to implement the solution.

LANGUAGE FEATURES STRATEGY AREA

The purpose of the language features strategy area is two-fold: First, to derive a good design to a programming problem you must know what features the programming language supports and how it provides them. Second, you may be forced by a particular programming project to use language features you've never used before. It can be daunting to have lots of requirements thrown at you in one project. The complexities associated with learning the Java language, learning how to create Java projects, learning an integrated development environment, and learning the process of solving a problem with a computer can induce panic. Use the language features strategy area to overcome this problem and maintain a sense of forward momentum.

Apply this strategy area by making a list of all the language features you need to study before starting your design. As you study each language feature mark it off your list. Take notes about each language feature and how it can be applied to your particular problem.

Table 3-5 presents an example check-off list for the language features used in the robot rat project.

Check-Off	Feature	Considerations
	Java Applications	How do you write a Java application? What's the purpose of the main() method. What is the meaning of the keywords public, static, and void? What code should go into the main() method? How do you run a Java application?
	classes	How do you declare and implement a class? What's the structure of a class. How do you create and compile a Java source file?

Table 3-5: Language Feature Study Check-Off List For Robot Rat Project

Check-Off	Feature	Considerations
	primitive data types	What is a primitive data type? How many are there? What is each one used for? How do you use them in a program? What range of values can each data type contain? How do you declare and use primitive data type variables or constants in a program?
	reference data types	What is a reference data type? How do you declare and use reference data types in a program? What's the purpose of the <i>new</i> operator? What pre-existing Java classes can be used to help create the program?
	array data types	What's the purpose and use of an array? How do you declare an array reference and use it in a program? What special functionality do array objects have?
	two-dimensional arrays	What is the difference between a two-dimensional array and a one-dimensional array? How do you declare and initialize a two-dimensional array? How do you access each element in a two-dimensional array?
	fields	What is a field? How do you declare class and instance fields? What's the difference between class fields and instance fields? What is the scope of a field?
	methods	What is a method? What are they good for? How do you declare and call a method? What's the difference between a static method and a non-static method? What are method parameters? How do you pass arguments to methods? How do you return values from methods?
	local variables	What is a local variable? How does their use affect class or instance fields? How long does a local variable exist? What is the scope of a local variable?
	constructor methods	What is the purpose of a constructor method? Can there be more than one constructor method? What makes constructor methods different from ordinary methods?
	flow-control statements	What is a flow-control statement? How do you use if, if/else, while, do, for, and switch statements in a program? What's the difference between for, while, and do? What's the difference between if, if/else, and switch?
	console I/O	What is console input and output? How do you print text to the console? How do you read text from the console and use it in your program?

Table 3-5: Language Feature Study Check-Off List For Robot Rat Project

Armed with your list of language features you can now study each one, marking it off as you go. When you come across a good code example that shows you how to use a particular language feature, copy it down or print it out and save it in a notebook for future reference.

Learning to program is a lot like learning to play a musical instrument. It takes observation and practice. You must put your trust in the masters and mimic their style. You may not at first fully understand why a particular piece of code works the way it does, or why they wrote it the way they did. But you will copy their style until you start to understand the underlying principles. Doing this builds confidence — slowly but surely. Soon you will have the skills required to set out on your own and write code with no help at all. In time, your programming skills will surpass those of your teachers.

After you have compiled and studied your list of language features you should just have a sense of what you can do with each feature and how to start the design process. More importantly, you will now know where to refer when you need to study a particular language feature in more depth. However, by no means will you have mastered the use of these features. So don't feel discouraged if, having arrived at this point, you still feel a bit overwhelmed by all that you must know. I must emphasize here that to master the art of programming takes practice, practice, practice!

Once you have studied each required language feature, you are ready to move on to the design strategy area of the project-approach strategy.

DESIGN STRATEGY AREA

Before you can solve the robot rat problem you must derive a plan of attack! The plan will consist of two essential elements: a high-level *software architecture diagram* and an *implementation approach*.

High-Level Software-Architecture Diagram

A high-level software-architecture diagram is a picture of the software components needed to implement the solution and their relationship to each other. Creating the high-level software-architecture diagram for the robot rat project is easy as the application will contain only one class. Complex projects, on the other hand, usually require many different classes, and each of these classes may interact with the others in some way. For these types of projects software-architecture diagrams play a key role in helping software engineers understand how the application works.

The Unified Modeling Language (UML) is used industry-wide to model software architectures. The UML class diagram for the RobotRat class at this early stage of project's design will look similar to figure 3-5.

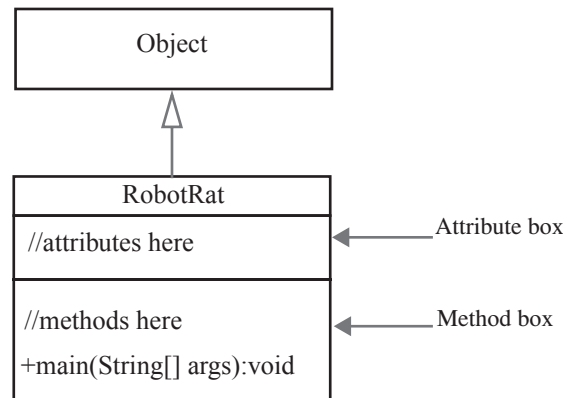


Figure 3-5: RobotRat UML Class Diagram

As figure 3-5 illustrates, the RobotRat class will extend (*inherit*) the functionality provided by the `java.lang.Object` class. This is indicated by the hollow-pointed arrow pointing from RobotRat to Object. (*In Java, all user-defined classes implicitly extend Object so you don't have to do anything special to achieve this functionality.*) The RobotRat class will have attributes (*fields*) and *methods*. Attributes are listed in the attribute box and methods are listed in the method box. The `main()` method is shown. The plus sign appearing to the left of the `main()` method indicates that it is a *public* method.

IMPLEMENTATION APPROACH

Before you begin coding you must have some idea of how you are going to translate the design into a finished project. Essentially, you must answer the following question: "Where do I start?" Getting started is ninety percent of the battle!

Generally speaking, when formulating an implementation approach you can proceed *macro-to-micro*, *micro-to-macro*, or a combination of both. (*I realize this sounds like unorthodox terminology but bear with me.*)

If you use the macro-to-micro approach, you build and test a code *framework* to which you incrementally add functionality that ultimately results in a finished project. If you use the micro-to-macro approach, you build and test small pieces of functionality first and then, bit-by-bit, combine them into a finished project.

More often than not you will use a combination of these approaches. Object-oriented design begs for macro-to-micro as a guiding approach but both approaches play well with each other as you will soon see. Java forces the issue somewhat by requiring that all methods and attributes belong to a class. (*Unlike C or C++ where functions, variables, and constants can exist independently.*)

When you start your design there will be many unknowns. For instance, you could attempt to specify all the methods required for the RobotRat class up front, but as you progress through development you will surely see the need for a method you didn't initially envision.

The following general steps outline a viable implementation approach to the robot rat project:

- Proceed from macro-to-micro by first creating and testing the RobotRat application class, devoid of any real functionality.
- Add and test a menu display capability.
- Add and test a menu-command processing framework by creating several empty methods that will serve as placeholders for future functionality. These methods are known as *method stubs*. (*Method stubbing is a great programming trick!*)
- Once the menu-command processing framework is tested, you must implement each menu item’s functionality. This means that the stub methods created in the previous step must be completely implemented and tested. The robot rat project is complete when all required functionality has been implemented and successfully tested.
- Develop the project iteratively. This means that you will execute the *plan-code-test-integrate* cycle many times on small pieces of the project until the project is complete.

Now that you have an overall implementation strategy you can proceed to the development cycle. The following sections walk you step-by-step through the iterative application of the development cycle.

DEVELOPMENT CYCLE FIRST ITERATION

Armed with an understanding of the project requirements, problem domain, language features, and an implementation approach, you are ready to begin development. To complete the project, apply the development cycle iteratively, meaning apply each of the development cycle phases —*plan*, *code*, *test*, and *integrate* —to a small, selected piece of the overall problem. When you’ve finished that piece of the project, select another piece and repeat the process. The following sections step through the iterative application of the development cycle to the robot rat project.

PLAN (FIRST ITERATION)

A good way to start each iteration of the development cycle is to list those pieces of the programming problem you are going to solve this time around. The list should have two columns: one that lists each piece of the program design or feature under consideration and one that notes the design decision made regarding that feature. Again, the purpose of the list is to help you maintain a sense of forward momentum. You may find, after you make the list, that you need more study in a particular language feature before proceeding to the coding step. That’s normal. Even seasoned programmers occasionally need to brush-up on unfamiliar or forgotten language features or application programming interfaces (API). (*i.e.*, *the Java platform APIs or third-party APIs*)

The list of the first pieces of the robot rat project that should be solved based on the previously discussed implementation approach is shown in table 3-6.

Check-Off	Design Consideration	Design Decision
	Program structure	One class will contain all the functionality.
	Creating the Java application class	The class name will be RobotRat. It will contain a “public static void main(String[] args){ }” method.
	constructor method	Write a class constructor that will print a short message to the screen when a RobotRat object is created.

Table 3-6: First Iteration Design Considerations

This is good for now. Although it doesn’t look like much, creating the RobotRat application class and writing a constructor that prints a short text message to the console are huge steps. You can now take this list and move to the code phase.

CODE (FIRST ITERATION)

Using your development environment create the RobotRat project. Create a Java source file named RobotRat.java and in it create the RobotRat class definition. To this class add the main() method and the RobotRat constructor method. When complete, your RobotRat.java source file should look similar to example 3.1.

3.1 RobotRat.java
(1st Iteration)

```

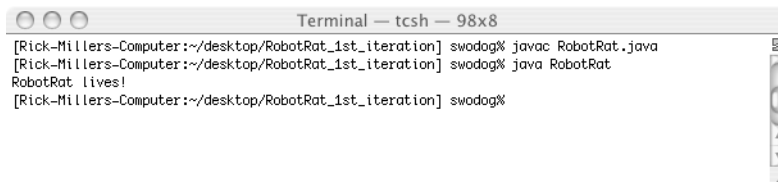
1      public class RobotRat {
2
3          public RobotRat() {
4              System.out.println("RobotRat lives!");
5          }
6
7
8          public static void main(String[] args) {
9              RobotRat rr = new RobotRat();
10             }
11         }

```

After you have created the source file you can move to the test phase.

TEST (FIRST ITERATION)

The test phase of the first iteration involves compiling the RobotRat.java file and running the RobotRat.class file. If the compilation results in errors you must return to the code phase, edit the file to make the necessary correction, and then attempt to compile and test again, repeating the cycle until you are successful. When you have successfully compiled and tested the first iteration of the RobotRat program you can move on to the next step of the development cycle. Figure 3-6 shows the results of running example 3.1.



```

Terminal — tcsh — 98x8
[Rick-Millers-Computer:~/desktop/RobotRat_1st_iteration] swodog% javac RobotRat.java
[Rick-Millers-Computer:~/desktop/RobotRat_1st_iteration] swodog% java RobotRat
RobotRat lives!
[Rick-Millers-Computer:~/desktop/RobotRat_1st_iteration] swodog%

```

Figure 3-6: Compiling & Testing RobotRat Class - First Iteration

INTEGRATE/TEST (FIRST ITERATION)

There's not a whole lot to integrate at this point so you are essentially done with the first development cycle iteration. Since this version of the robot rat project is contained in one class named RobotRat, any changes you make directly to the source file are immediately integrated. However, for larger projects, you will want to code and test a piece of functionality, usually at the class granularity, before adding the class to the larger project.

You are now ready to move to the second iteration of the development cycle.

DEVELOPMENT CYCLE SECOND ITERATION

With the RobotRat application class structure in place and tested, it is time to add another piece of functionality to the program.

PLAN (SECOND ITERATION)

A good piece of functionality to add to the RobotRat class at this time would be the text menu that displays the list of available robot rat control options. Refer to the project specification to see a suggested menu example. Table 3-7 lists the features that will be added to RobotRat during this iteration.

Check-Off	Design Consideration	Design Decision
	Display control menu to the console	Create a RobotRat method named printMenu() that prints the menu to the console. Test the printMenu() method by calling it from the constructor. The printMenu() method will return void and take no arguments. It will use the System.out.println() method to write the menu text to the console.

Table 3-7: Second Iteration Design Considerations

This looks like enough work to do for one iteration. You can now move to the code phase.

CODE (SECOND ITERATION)

Edit the RobotRat.java source file to add a method named printMenu(). Example 3.2 shows the RobotRat.java file with the printMenu() method added.

3.2 RobotRat.java
(2nd Iteration)

```

1      public class RobotRat {
2
3          public RobotRat(){
4              printMenu();
5          }
6
7          public void printMenu(){
8              System.out.println("\n\n");
9              System.out.println("    Robot Rat Control Menu");
10             System.out.println("\n\n");
11             System.out.println("  1. Pen Up");
12             System.out.println("  2. Pen Down");
13             System.out.println("  3. Turn Right");
14             System.out.println("  4. Turn Left");
15             System.out.println("  5. Move Forward");
16             System.out.println("  6. Print Floor");
17             System.out.println("  7. Exit");
18             System.out.println("\n\n\n");
19         }
20
21
22         public static void main(String[] args){
23             RobotRat rr = new RobotRat();
24         }
25     }

```

The printMenu() method begins on line 7. Notice how it's using the System.out.println() method to write menu text to the console. The "\n" is the escape sequence for the newline character. Several of these are used to add line spacing as an aid to menu readability.

Notice also that the code in the constructor has changed. The line printing the "RobotRat Lives!" message was removed and replaced by a call to the printMenu() method on line 4.

When you've finished making the edits to RobotRat.java you are ready to compile and test.

TEST (SECOND ITERATION)

Figure 3-7 shows the results of testing RobotRat at this stage of development. The menu seems to print well enough.

```

Terminal — tcsh — 99x19
[Rick-Millers-Computer:~/desktop/RobotRat_2nd_Iteration] swodog% java RobotRat

Robot Rat Control Menu

1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move Forward
6. Print Floor
7. Exit

[Rick-Millers-Computer:~/desktop/RobotRat_2nd_Iteration] swodog% 

```

Figure 3-7: Compiling & Testing RobotRat.class - Second Iteration

INTEGRATE/TEST (SECOND ITERATION)

Again, there is not much to integrate or regression test. If you are happy with the way the menu looks on the console you can move on to the third iteration of the development cycle.

DEVELOPMENT Cycle Third Iteration

OK. The class structure is in place and the menu is printing to the screen. The next piece of the project to work on should be the processing of menu commands. Here's where you can employ the technique of method stubbing so you can worry about the details later.

PLAN (Third Iteration)

Table 3.8 lists the design considerations for this iteration.

Check-Off	Design Consideration	Design Decision
	Read the user's desired menu command from the console	<p>The <code>System.out.println()</code> method makes it easy to write text to the console but you'll have to build a method that reads text from the console. The <code>System.in</code> object is an <code>InputStream</code> object. This <code>InputStream</code> object can be used to create an <code>InputStreamReader</code> object, which can be used to create a <code>BufferedReader</code> object. The <code>InputStreamReader</code> and <code>BufferedReader</code> classes are found in the <code>java.io</code> package.</p> <p>Once you get the text string from the console you will only want to use the first letter of the string. The <code>String</code> class provides a <code>charAt()</code> method that returns the character located at some index location in the string.</p> <p>Write a user-defined method named <code>readChar()</code> that puts these pieces together to read the user's input as a string and return the first character.</p>

Table 3-8: Third Iteration Design Considerations

Check-Off	Design Consideration	Design Decision
	Execute the user's selected menu command	<p>This will be accomplished with a user-defined method named <code>processMenuChoice()</code>. This method will return void and take no arguments.</p> <p>The body of the <code>processMenuChoice()</code> method will utilize a switch statement that acts on the menu command entered by the user. Each case of the switch statement will call a user-defined method to execute the required functionality. These methods will be named: <code>setPenUp()</code>, <code>setPenDown()</code>, <code>turnLeft()</code>, <code>turnRight()</code>, <code>move()</code>, <code>printFloor()</code>, and <code>exit()</code>.</p>
	Use method stubbing to test <code>processMenuChoice()</code> method	<p>The user-defined methods mentioned above with the exception of <code>exit()</code> will be stubbed out. The only functionality they will provide will be to print a short test message to the console.</p> <p>The <code>exit()</code> method will call the <code>System.exit()</code> method to exit the program.</p>

Table 3-8: Third Iteration Design Considerations

This looks like it will keep you busy for a while. You will actually move between the code and test phase repeatedly in this iteration, almost as if it were a mini development spiral in and of itself. The best place to start is at the top of the list.

CODE (THIRD ITERATION)

Starting with the first feature on the list, reading the user's command from the console, you will need to import the `java.io` package into the `RobotRat.java` file so you don't have to fully qualify the names of the `InputStreamReader` and `BufferedReader` classes.

Example 3.3 gives the code for the `RobotRat.java` file with the `readChar()` and `processMenuChoice()` methods completed.

3.3 *RobotRat.java*
(3rd Iteration)

```

1      import java.io.*;
2
3      public class RobotRat {
4          /**
5           * private attributes
6           */
7          private BufferedReader console = null;
8
9          public RobotRat(){
10             //Initialize RobotRat attributes
11             console = new BufferedReader(new InputStreamReader(System.in));
12         }
13
14         public void run(){
15             while(true){
16                 printMenu();
17                 processMenuChoice();
18             }
19         }
20
21         public void printMenu(){
22             System.out.println("\n\n");
23             System.out.println("    Robot Rat Control Menu");
24             System.out.println("\n");
25             System.out.println("  1. Pen Up");
26             System.out.println("  2. Pen Down");
27             System.out.println("  3. Turn Right");
28             System.out.println("  4. Turn Left");
29             System.out.println("  5. Move Forward");
30             System.out.println("  6. Print Floor");
31             System.out.println("  7. Exit");
32             System.out.println("\n\n");
33         }
34     }

```



```

35     public char readChar(){
36         String s = null;
37         System.out.print("Please select from the menu: ");
38         try{
39             s = console.readLine();
40         }catch(Exception ignored){}
41         return s.charAt(0);
42     }
43
44     public void processMenuChoice(){
45
46         switch(readChar()){
47             case '1': setPenUp();
48                 break;
49             case '2': setPenDown();
50                 break;
51             case '3': turnRight();
52                 break;
53             case '4': turnLeft();
54                 break;
55             case '5': move();
56                 break;
57             case '6': printFloor();
58                 break;
59             case '7': exit();
60             default: printErrorMessage();
61         }
62     }
63
64     public void setPenUp(){System.out.println("SetPenUp() method");}
65     public void setPenDown(){System.out.println("SetPenDown() method");}
66     public void turnRight(){System.out.println("turnRight() method");}
67     public void turnLeft(){System.out.println("turnLeft() method");}
68     public void move(){System.out.println("move() method");}
69     public void printFloor(){System.out.println("printFloor() method");}
70
71     public void exit(){
72         System.exit(0);
73     }
74
75     public void printErrorMessage(){
76         System.out.println("Please enter a valid menu choice!");
77     }
78
79     public static void main(String[] args){
80         RobotRat rr = new RobotRat();
81         rr.run();
82     }
83 }

```

There are a couple of important points to note in the code example above. First, the `readChar()` and `processMenuChoice()` methods have been added. The stub methods appear on lines 64 through 69. Each stub method prints a short message to the screen giving the name of the method called. The `exit()` method appears on line 71 and uses the `System.exit()` method to do its work.

There are two additional utility methods: `printErrorMessage()` which starts on line 75, and the `run()` method that begins on line 14. The `printErrorMessage()` method is called by the default case of the switch statement located in the `processMenuChoice()` method. It's a good idea to give the user some feedback when they enter an invalid menu choice. This is an example of the type of functionality you'll add to your programs although it is not specifically called for in a project specification.

The `run()` method is used to get the `RobotRat` program going. It repeatedly loops, with the help of a forever-repeating while statement, displaying the menu and processing user commands. The program will run until the user chooses the Exit menu item.

Notice that another line of code has been added to the `main()` method on line 81. All the `main()` method does is create a `RobotRat` object and assign it to the reference `rr`. Then, on line 81 the `run()` method is called via the `rr` reference.

Public vs. Private Methods

Since all the methods of RobotRat, with the exception of the constructor, run(), and main() methods, are called internally to the RobotRat class, they could be declared private. The constructor, run(), and main() methods are declared public, although technically, only the main() method must be public in this application.

TEST (THIRD ITERATION)

The test phase will consist of running the revised RobotRat program and testing the menu commands. Each menu command entered should result in the stub method message being printed to the screen. An invalid command should result in the error message. Figure 3-8 shows a screen shot of a partial test of the RobotRat menu commands.

INCREMENTAL TESTING

Although you could have tried to make all the required modifications to the RobotRat class at one stroke you most likely would make small changes or additions to the code and then immediately compile and test the results. For instance, you could concentrate on writing the readChar() method and test it exclusively before coding the processMenuChoice() method.

Another area where proceeding in small steps would be a good idea would be the body of the processMenuChoice() method. The switch statement could have been written one case at a time. Then, once you gain confidence that your code will work as planned you can code up the rest in short order.

INTEGRATE/TEST (THIRD ITERATION)

Now that you are accepting and processing user menu commands you can examine the effect this has on the menu display. If you look at figure 3-8 it appears as if there might be too many spaces between the last menu item and the menu entry prompt. Thus, after the addition of a feature you've noted its effects on another program feature. The space issue can be adjusted in the next development cycle iteration.



Figure 3-8: Testing Menu Commands

DEVELOPMENT CYCLE FOURTH ITERATION

The RobotRat code framework is in place. Users can run the program, select menu choices, and see the results of their actions via stub method console messages. It's now time to start adding detailed functionality to the RobotRat class.

PLAN (FOURTH ITERATION)

It's now time to start adding more attributes to the RobotRat class and manipulating those attributes. Two good attributes to start with are the robot rat's direction and pen_position. Another good piece of functionality to implement is the floor. A good goal would be to create, initialize, and print the floor. It would also be nice to load the floor with one or more test patterns.

Table 3-9 lists the design considerations for this development cycle iteration. As you will soon see, more detailed analysis is required to implement the selected robot rat program features. This analysis may require the use of design drawings such as flow charts, state-transition diagrams, and class diagrams, in addition to pseudocode, and other design techniques.

Check-Off	Design Consideration	Design Decision
	Implement Robot Rat's direction	<p>direction will be an integer (int) variable. It will have four possible states or values: NORTH, SOUTH, EAST, and WEST. You can implement these as class constants. This will make the source code easier to read and maintain.</p> <p>The robot rat's direction will change when either the turnLeft() or turnRight() methods are called.</p> <p>The initial direction upon program startup will be EAST.</p>
	Implement pen_position	<p>pen_position will be an integer variable. It will have two valid states or values: UP and DOWN. These can be implemented as class constants as well. The robot rat's pen_position will change when either the setPenUp() or the setPenDown() methods are called.</p> <p>The initial pen_position value will be UP upon program startup.</p>
	floor	<p>The floor will be a two-dimensional array of boolean variables. If an element of the floor array is set to true it will result in the '*' character being printed to the console. If an element is false the '0' character will be printed to the console.</p> <p>The floor array elements will be initialized to false upon program startup.</p>

Table 3-9: Fourth Iteration Design Considerations

This will keep you busy for a while. You might have to spend some more time analyzing the issues regarding setting the robot rat's direction and its pen_position. It is often helpful to draw state transition diagrams to graphically illustrate state changes to objects. Figure 3-9 shows the state transition diagram for pen_position.

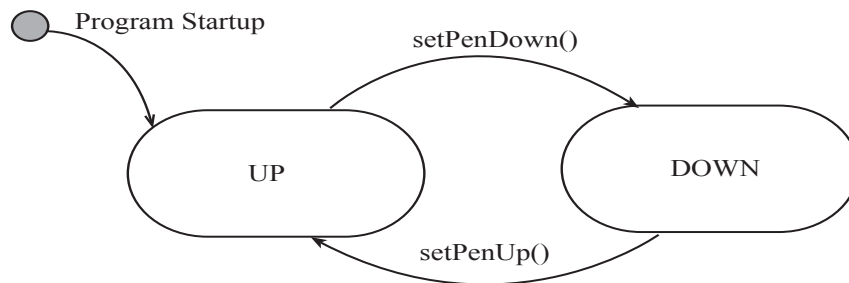


Figure 3-9: pen_position State Transition Diagram

As figure 3-9 illustrates, the pen_position variable is set to the UP state upon program startup. It will remain UP until the setPenDown() method is called, at which time it will be set to the DOWN state. A similar state transition diagram is shown for the direction variable in figure 3-10.

As is illustrated in figure 3-10, the robot rat's direction is initialized to EAST upon program startup. Each call to the turnLeft() or turnRight() methods will change the state (*value*) of the direction variable.

IMPLEMENTING STATE TRANSITION DIAGRAMS

State transition diagrams of this nature are easily implemented using a switch statement. (*You could use an if/else statement but the switch works well in this case*) Example 3.4 gives a pseudocode description for the turnLeft() method:

```

1      check the value of the direction variable
2      if direction equals EAST then set the value of direction to NORTH
3      else if direction equals NORTH then set the value of direction to WEST
4      else if direction equals WEST then set the value of direction to SOUTH
5      else if direction equals SOUTH then set the value of direction to EAST
6      else if direction equals an invalid state set the value of direction to EAST.
  
```

3.4 Pseudocode for turnLeft() Method

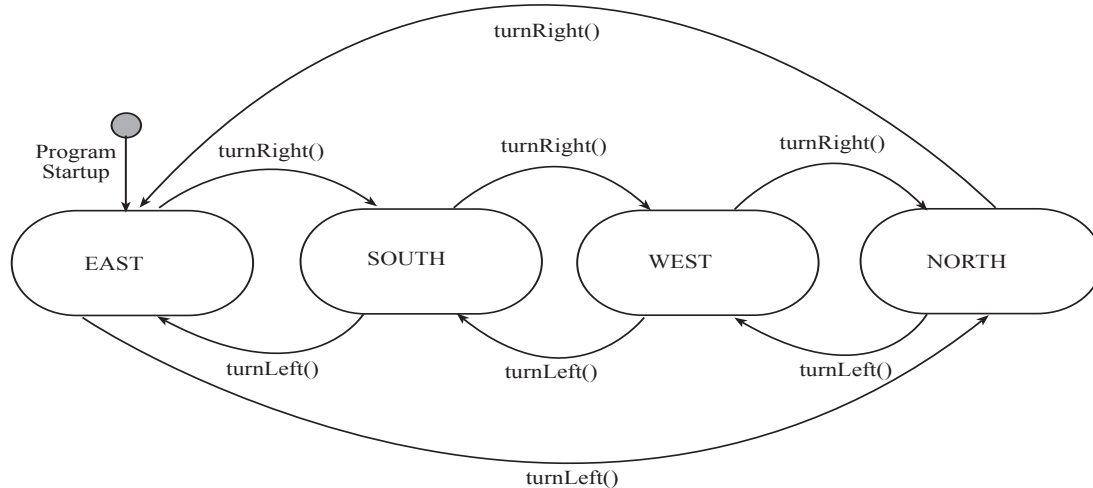


Figure 3-10: State Transition Diagram for the direction Variable

You could construct a pseudocode description for the `turnRight()`, `setPenUp()`, and `setPenDown()` methods as well, using the state transition diagrams as a guide.

IMPLEMENTING THE `PRINTFLOOR()` METHOD

Example 3.5 gives the pseudocode for the `printFloor()` method.

3.5 Pseudocode for `printFloor()` Method

```

1   for each row in the floor do the following
2     for each column in each row do the following
3       check the value of each floor element
4       if the value is true print the '*' character to the console
5       else if the value is false print the '0' character to the console
6       print a newline character at the end of each row
  
```

When you feel you have done enough analysis of the current set of `RobotRat` features you can move to the code phase of development cycle.

CODE (FOURTH ITERATION)

The `printFloor()`, `turnLeft()`, `turnRight()`, `setPenUp()`, and `setPenDown()` methods already exist as stub methods. You will proceed in this phase to add the required code to each of these methods, then compile and test the results. Just like the previous iteration, this code phase will comprise multiple code, compile, and test cycles.

To proceed you must first add the `RobotRat` field declarations at the top of the class. Any fields declared here should be initialized in the constructor method. The floor is initialized with a special method named `initializeFloor()`. Example 3.6 shows a partial code listing for this section of the `RobotRat.java` source file along with the constructor method.

*3.6 RobotRat.java
(4th Iteration Partial Listing)*

```

1   import java.io.*;
2
3   public class RobotRat {
4     /**
5     * private instance attributes
6     */
7     private BufferedReader console = null;
8     private int pen_position = 0;
9     private int direction = 0;
10    private boolean floor[][] = null;
11
12    /**
13    * class constants
14    */
15    private static final int NORTH = 0;
16    private static final int SOUTH = 1;
  
```

```

17     private static final int EAST = 2;
18     private static final int WEST = 3;
19     private static final int UP = 0;
20     private static final int DOWN = 1;
21
22
23     public RobotRat(int rows, int cols){
24         //Initialize RobotRat attributes
25         console = new BufferedReader(new InputStreamReader(System.in));
26         direction = EAST;
27         pen_position = UP;
28         floor = new boolean[rows][cols];
29         initializeFloor();
30     }
31
32     private void initializeFloor(){
33         for(int i = 0; i<floor.length; i++){
34             for(int j = 0; j<floor[i].length; j++){
35                 floor[i][j] = false;
36             }
37         }
38     }
39

```

There are a few items to note with this code example. First, in addition to the `pen_position`, `direction`, and `floor` fields, the class constants were added as well. More field initialization code was added to the constructor method as well as two new method parameters named `rows` and `cols`. The purpose of the constructor parameters will be to initialize the size of the floor array when the `RobotRat` object is created in the `main()` method.

The `initializeFloor()` method is defined beginning on line 32. It simply sets each element of the floor array to false. Notice that the `initializeFloor()` method is declared to be private since it is only intended to be used internally by the `RobotRat` class.

It's now time to turn your attention to the `setPenUp()`, `setPenDown()`, `turnLeft()`, and `turnRight()` methods.

Example 3.7 shows the source code for the `setPenUp()` method.

3.7 setPenUp() method

```

1     private void setPenUp(){
2         pen_position = UP;
3         System.out.println("The pen_position is UP");
4     }

```

As you can see it's fairly straightforward. It just sets the `pen_position` attribute to the UP state and then prints a short message to the console saying the `pen_position` is UP. Example 3.8 gives the code for the `setPenDown()` method.

3.8 setPenDown() method

```

1     private void setPenDown(){
2         pen_position = DOWN;
3         System.out.println("pen_position is DOWN");
4     }

```

This method is similar to the previous method only it's setting the `pen_position` to the opposite state. Let's look now at the `turnLeft()` method as shown in example 3.9.

3.9 turnLeft() method

```

1     private void turnLeft(){
2         switch(direction){
3             case NORTH: direction = WEST;
4                 System.out.println("RobotRat facing WEST");
5                 break;
6             case EAST: direction = NORTH;
7                 System.out.println("RobotRat facing NORTH");
8                 break;
9             case SOUTH: direction = EAST;
10                System.out.println("RobotRat facing EAST");
11                break;
12             case WEST: direction = SOUTH;
13                System.out.println("RobotRat facing SOUTH");
14                break;
15             default: direction = EAST;
16                System.out.println("RobotRat facing EAST");
17         }
18     }

```

Notice in the `turnLeft()` method above that the switch statement checks the value of the direction field then executes the appropriate case statement. The `turnRight()` method is coded in similar fashion using the state transition diagram as a guide.

The `printFloor()` method is all that's left for this iteration and is shown in example 3.10.

3.10 printFloor() method

```

1     private void printFloor(){
2         for(int i = 0; i<floor.length; i++){
3             for(int j=0; j<floor[i].length; j++){
4                 if(floor[i][j] == true)
5                     System.out.print('*');
6                 else System.out.print('0');
7             }
8             System.out.println();
9         }
10    }

```

TEST (FOURTH ITERATION)

There's a lot to test for this iteration. You'll need to test all the methods that were modified and you'll be anxious to test the `printFloor()` method since now you'll see the floor pattern print to the console. Figure 3-11 shows the `printFloor()` method being tested.

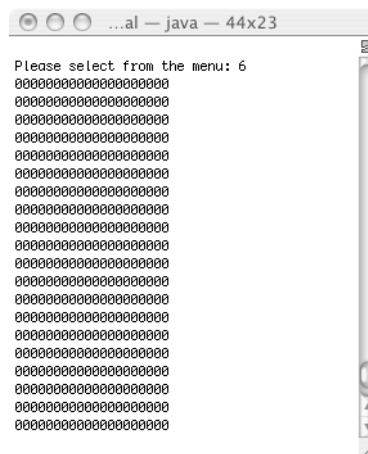


Figure 3-11: `printFloor()` Method Test

As you can see, it just prints the `'0'` characters to the screen. You might find it helpful to load the array with a test pattern to test the `'*'` characters as well.

INTEGRATE/TEST (FOURTH ITERATION)

Check to see how all the new functionality you added has affected any previously working functionality. It would also be a good idea to see how the floor looks using different floor array dimensions. The changes to the `RobotRat` constructor method makes it easy to create `RobotRat` objects with different floor sizes. When you are happy with all the work done in this iteration it is time to move on to the next development cycle iteration.

DEVELOPMENT CYCLE FIFTH ITERATION

All that's left to code is the `move()` method. All the other supporting pieces are now in place. You can change the robot rat's direction by turning left or right, and change the pen's position up or down. The `move()` method will use all this functionality to model the movement of the robot rat across the floor.

As you get into the planning phase of this iteration you may discover you need more than just the `move()` method. For instance, you'll need some way to get the number of spaces to move the robot rat from the user.

PLAN (Fifth Iteration)

Table 3-10 lists the design considerations for this iteration of the development cycle.

Check-Off	Design Consideration	Design Decision
	move method	Write the code for the <code>move()</code> method. The <code>move()</code> method will use the <code>direction</code> and <code>pen_position</code> fields to make move decisions. The <code>move()</code> method will need a way to get the number of spaces to move from the user. You will have to add the <code>current_row</code> and <code>current_col</code> fields to the <code>RobotRat</code> class. These fields will be used to preserve the robot rat's floor position information between moves.
	Getting the number of spaces to move from the user	When the user moves the robot rat they will need to enter the number of spaces so the move can be executed. This can be handled by writing a new method called <code>getSpaces()</code> . The <code>getSpaces()</code> method can be a slight modification of the <code>readChar()</code> method as it will read a text string from the user and convert it to an integer. The <code>getSpaces()</code> method will return an integer value and take no parameters.
	Add <code>current_row</code> & <code>current_col</code> fields	<code>current_row</code> and <code>current_col</code> will be declared with the rest of the <code>RobotRat</code> fields and initialized in the constructor.

Table 3-10: Fifth Iteration Design Considerations

The first feature from the list to be coded and tested should be the `getSpaces()` method. The `readChar()` method provides an example of some of the functionality you'll require, namely, read a text string from the console. This text string must then be converted into an integer value. Java provides a set of helper classes that will perform the conversion for you. In this instance you'll need to use the services of the `java.lang.Integer` class.

The `move()` method will require you to do some intense study of the mechanics of a robot rat move. A picture like figure 3.3 is very helpful in this particular case because it enables you to work out the moves of a robot rat upon the floor. You should work out the moves along the north, south, east, and west directions and note how to execute a move in terms of the robot rat's `current_row`, `current_col`, and `direction` fields.

It's a good idea to place the robot rat in a starting position. A good starting position is on `current_row = 0`, `current_col = 0`, and `direction = EAST`. These attributes can be initialized to the required values or states in the `RobotRat` constructor method.

Once you get the move mechanics worked out you can write a pseudocode description of the `move()` method like the one presented below in example 3.11.

3.11 `move()` method pseudocode

```

1      get spaces to move from user
1      if pen_position is UP do the following
1          if direction is NORTH move RobotRat NORTH -- do not mark floor
1          if direction is SOUTH move RobotRat SOUTH -- do not mark floor
1          if direction is EAST move RobotRat EAST -- do not mark floor
1          if direction is WEST move RobotRat WEST -- do not mark floor
1      if pen_position is DOWN
1          if direction is NORTH move RobotRat NORTH -- mark floor
1          if direction is SOUTH move RobotRat SOUTH -- mark floor
1          if direction is EAST move RobotRat EAST -- mark floor
1          if direction is WEST move RobotRat WEST -- mark floor

```

With your feature planning complete you can move to the code phase.

CODE (FIFTH ITERATION)

Example 3.12 gives the source code for the `getSpaces()` method.

3.12 `getSpaces()` method

```

1     private int getSpaces(){
2         int temp = 0;
3         try{
4             temp = Integer.parseInt(console.readLine());
5         }catch(Exception e){
6             System.out.println("Spaces has been set to zero!");
7         }
8         return temp;
9     }

```

The `getSpaces()` method utilizes the `Integer` wrapper class's `parseInt()` method on line 4 along with the `console.readLine()` method to read a text string from the console and convert it into an integer value. If the `parseInt()` method throws an exception a message is sent to the console stating the value of spaces has been set to zero and the `temp` variable is returned. The `getSpaces()` method can now be used in the `move()` method as is shown in example 3.13.

3.13 `move()` method

```

1     private void move(){
2         System.out.print("Please enter spaces to move: ");
3         int spaces = getSpaces();
4         switch(pen_position){
5             case UP: switch(direction){
6                 case NORTH: if((current_row - spaces) <= 0)
7                             current_row = 0;
8                             else current_row = current_row - spaces;
9                             break;
10                case SOUTH: if((current_row + spaces) >= (floor[0].length - 1))
11                             current_row = (floor[0].length - 1);
12                             else current_row = current_row + spaces;
13                             break;
14                case EAST: if((current_col + spaces) >= (floor.length - 1))
15                             current_col = (floor.length - 1);
16                             else current_col = current_col + spaces;
17                             break;
18                case WEST: if((current_col - spaces) <= 0)
19                             current_col = 0;
20                             else current_col = current_col - spaces;
21                             break;
22            }
23            break;
24            case DOWN: switch(direction){
25                case NORTH: if((current_row - spaces) <= 0){
26                            while(current_row >= 1)
27                                floor[current_row--][current_col] = true;
28                            }
29                            else while(spaces-- > 0)
30                                floor[current_row--][current_col] = true;
31                            break;
32                case SOUTH: if((current_row + spaces) >= (floor[0].length - 1)){
33                            while(current_row < (floor[0].length-1))
34                                floor[current_row++][current_col] = true;
35                            }
36                            else while(spaces-- > 0)
37                                floor[current_row++][current_col] = true;
38                            break;
39                case EAST: if((current_col + spaces) >= (floor.length - 1)){
40                            while(current_col < (floor.length-1))
41                                floor[current_row][current_col++] = true;
42                            }
43                            else while(spaces-- > 0)
44                                floor[current_row][current_col++] = true;
45                            break;
46                case WEST: if((current_col - spaces) <= 0){
47                            while(current_col >= 1)
48                                floor[current_row][current_col--] = true;
49                            }
50                            else while(spaces-- > 0)
51                                floor[current_row][current_col--] = true;
52                            break;
53            }
54            break;
55        }
56    }

```


TEST (Fifth Iteration)

This will be the most extensive test session of the RobotRat project yet. You must test the `move()` method in all directions and ensure you are properly handling move requests that attempt to go beyond the bounds of the floor array.

Figure 3-12 shows the RobotRat project being run and a user entering a number of spaces to move. Figure 3-13 shows two floor patterns printed to the console.

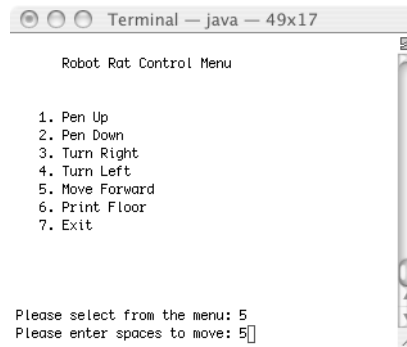


Figure 3-12: Testing the `getSpaces()` and `move()` Methods

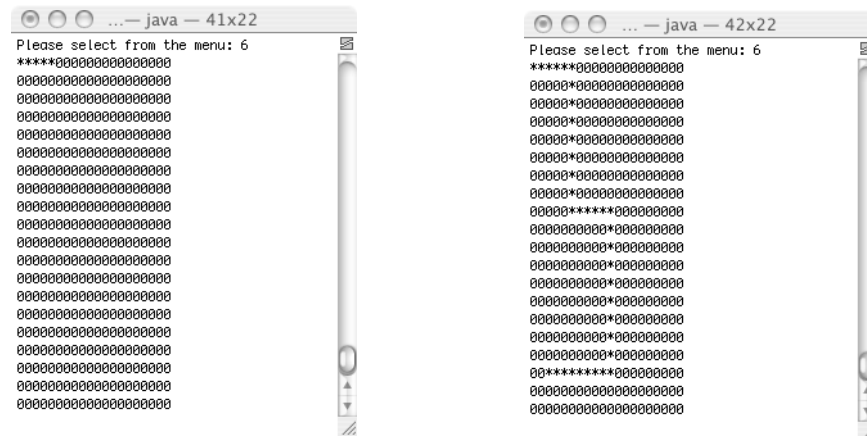


Figure 3-13: Two Floor Patterns Printed To The Console

INTEGRATE/TEST (Fifth Iteration)

At this point in the development cycle you will want to test the entire RobotRat project. Move the robot rat with the pen up and pen down. Move in all directions and try making and printing different floor patterns. The emphasis in this phase is to test all RobotRat functionality and note the effects the addition of the latest features had on existing functionality.

SOME FINAL CONSIDERATIONS

The core development efforts are complete, but I would hesitate in calling RobotRat finished just yet. When you have finished the last integration and testing cycle you will want to revisit and edit the code for neatness and clarity. Perhaps you can do a better job of formatting the code so it's easier to read. Maybe you thought of a few cool features to add to the project to get extra credit.

When you have reached the point where you feel you are done with the project you will want to give it one last review. Table 3-11 lists a few things to check before submitting your project to your instructor.

Check-Off	Double-Check	To ensure...
	Source Code Formatting	Ensure it is neat, consistently aligned, and indented to aid readability
	Comments	Make sure they're used to explain critical parts of your code and that they are consistently formatted. Use javadoc comments if possible.
	File Comment Header	Add a file comment header at the top of all project source files. Make sure it has your name, class, instructor, and the name of the project. The file comment header format may be dictated by your instructor or by coding guidelines established at work.
	Printed copies of source code files	Make sure that it fits on a page in a way that preserves formatting and readability. Adjust the font size or paper orientation if required to ensure your project looks professional.
	Class files on floppy, CD-ROM, or memory stick (i.e., removable media)	Ensure all the required source and class files are present. Try running your project from the removable media to make sure you have included all the required class files.

Table 3-11: Final Project Review Check-List

Figure 3-14 shows a screen shot from the automatically generated javadoc documentation for the RobotRat class.

Method Summary	
private void	exit() Exits the RobotRat program
private int	getSpaces() Gets number of spaces to move the RobotRat
private void	initializeFloor() Initializes each floor array element to false
static void	main (java.lang.String[] args) The main method.
private void	move() This function moves the RobotRat about the floor.
private void	printErrorMessage() Prints error text message when invalid RobotRat menu item entered
private void	printFloor() Prints the floor array pattern to the console
private void	printMenu() Prints the RobotRat control menu to the console
private void	processMenuChoice() Menu processing method.
private char	readChar() Reads text string user input and returns the first character
void	run() Repeatedly displays the RobotRat control menu and processes user commands.
private void	setPenDown() Sets the RobotRat's pen to the DOWN position
private void	setPenUp() Sets the RobotRat's pen to the UP position
private void	turnLeft() Turns the RobotRat to the left
private void	turnRight() Turns the RobotRat to the right

Figure 3-14: Partial RobotRat Javadoc Documentation

COMPLETE ROBOTRAT.JAVA SOURCE CODE LISTING

3.14 RobotRat.java
(Complete Listing)

```

1      /* *****
2      Class:   IST 149
3      Project: Robot Rat
4      Student: Rick Miller
5      ***** */
6
7      import java.io.*;
8
9      /**
10     * @author <b> Rick Miller </b>
11     * @version 1.0
12     *
13     */
14     public class RobotRat {
15         // private instance attributes
16
17         private BufferedReader console = null;
18         private int pen_position = 0;
19         private int direction = 0;
20         private boolean floor[][] = null;
21         private int current_row = 0;
22         private int current_col = 0;
23
24         // class constants
25
26         private static final int NORTH = 0;
27         private static final int SOUTH = 1;
28         private static final int EAST = 2;
29         private static final int WEST = 3;
30         private static final int UP = 0;
31         private static final int DOWN = 1;
32
33         /**
34         * RobotRat constructor
35         * @param rows The number of rows to create in the floor array
36         * @param cols The number of columns to create in the floor array
37         */
38         public RobotRat(int rows, int cols){
39             //Initialize RobotRat attributes
40             console = new BufferedReader(new InputStreamReader(System.in));
41             direction = EAST;
42             pen_position = UP;
43             current_row = 0;
44             current_col = 0;
45             floor = new boolean[rows][cols];
46             initializeFloor();
47         }
48
49         /**
50         * Initializes each floor array element to false
51         */
52         private void initializeFloor(){
53             for(int i = 0; i<floor.length; i++){
54                 for(int j = 0; j<floor[i].length; j++){
55                     floor[i][j] = false;
56                 }
57             }
58         }
59
60         /**
61         * Repeatedly displays the RobotRat control menu and
62         * processes user commands.
63         */
64         public void run(){
65             while(true){
66                 printMenu();
67                 processMenuChoice();
68             }
69         }
70
71         /**
72         * Prints the RobotRat control menu to the console
73         */

```

```

74     private void printMenu(){
75         System.out.println("\n\n");
76         System.out.println("    Robot Rat Control Menu");
77         System.out.println("\n");
78         System.out.println("  1. Pen Up");
79         System.out.println("  2. Pen Down");
80         System.out.println("  3. Turn Right");
81         System.out.println("  4. Turn Left");
82         System.out.println("  5. Move Forward");
83         System.out.println("  6. Print Floor");
84         System.out.println("  7. Exit");
85         System.out.println("\n\n\n");
86     }
87
88     /**
89     * Gets number of spaces to move the RobotRat
90     * @return An integer value representing spaces to move
91     */
92     private int getSpaces(){
93         int temp = 0;
94         try{
95             temp = Integer.parseInt(console.readLine());
96         }catch(Exception e){
97             System.out.println("Spaces has been set to zero!");
98         }
99         return temp;
100    }
101
102    /**
103    * Reads text string user input and returns the first character
104    * @return First character of user text string input
105    */
106    private char readChar(){
107        String s = null;
108        System.out.print("Please select from the menu: ");
109        try{
110            s = console.readLine();
111        }catch(Exception ignored){}
112        return s.charAt(0);
113    }
114
115    /**
116    * Menu processing method.
117    * @see #readChar()
118    */
119    private void processMenuChoice(){
120
121        switch(readChar()){
122            case '1': setPenUp();
123                    break;
124            case '2': setPenDown();
125                    break;
126            case '3': turnRight();
127                    break;
128            case '4': turnLeft();
129                    break;
130            case '5': move();
131                    break;
132            case '6': printFloor();
133                    break;
134            case '7': exit();
135            default: printErrorMessage();
136        }
137    }
138
139    /**
140    * Sets the RobotRat's pen to the UP position
141    */
142    private void setPenUp(){
143        pen_position = UP;
144        System.out.println("The pen_position is UP");
145    }
146
147    /**
148    * Sets the RobotRat's pen to the DOWN position
149    */
150    private void setPenDown(){
151        pen_position = DOWN;
152        System.out.println("pen_position is DOWN");
153    }
154

```

```

155
156     /**
157     * Turns the RobotRat to the right
158     */
159     private void turnRight(){
160         switch(direction){
161             case NORTH: direction = EAST;
162                         System.out.println("RobotRat facing EAST");
163                         break;
164             case EAST:  direction = SOUTH;
165                         System.out.println("RobotRat facing SOUTH");
166                         break;
167             case SOUTH: direction = WEST;
168                         System.out.println("RobotRat facing WEST");
169                         break;
170             case WEST:  direction = NORTH;
171                         System.out.println("RobotRat facing NORTH");
172                         break;
173             default:    direction = EAST;
174                         System.out.println("RobotRat facing EAST");
175         }
176         System.out.println("turnRight() method");
177     }
178
179     /**
180     * Turns the RobotRat to the left
181     */
182     private void turnLeft(){
183         switch(direction){
184             case NORTH: direction = WEST;
185                         System.out.println("RobotRat facing WEST");
186                         break;
187             case EAST:  direction = NORTH;
188                         System.out.println("RobotRat facing NORTH");
189                         break;
190             case SOUTH: direction = EAST;
191                         System.out.println("RobotRat facing EAST");
192                         break;
193             case WEST:  direction = SOUTH;
194                         System.out.println("RobotRat facing SOUTH");
195                         break;
196             default:    direction = EAST;
197                         System.out.println("RobotRat facing EAST");
198         }
199     }
200
201     /**
202     * This method moves the RobotRat about the floor.
203     * @see #getSpaces()
204     */
205     private void move(){
206         System.out.print("Please enter spaces to move: ");
207         int spaces = getSpaces();
208         switch(pen_position){
209             case UP: switch(direction){
210                 case NORTH: if((current_row - spaces) <= 0)
211                             current_row = 0;
212                             else current_row = current_row - spaces;
213                             break;
214                 case SOUTH: if((current_row + spaces) >= (floor[0].length - 1))
215                             current_row = (floor[0].length - 1);
216                             else current_row = current_row + spaces;
217                             break;
218                 case EAST:  if((current_col + spaces) >= (floor.length - 1))
219                             current_col = (floor.length - 1);
220                             else current_col = current_col + spaces;
221                             break;
222                 case WEST:  if((current_col - spaces) <= 0)
223                             current_col = 0;
224                             else current_col = current_col - spaces;
225                             break;
226             }
227             break;
228             case DOWN: switch(direction){
229                 case NORTH: if((current_row - spaces) <= 0){
230                             while(current_row >= 1)
231                                 floor[current_row--][current_col] = true;
232                             }
233                 else while(spaces-- > 0)
234                     floor[current_row--][current_col] = true;
235                 break;

```

```

236         case SOUTH: if((current_row + spaces) >= (floor[0].length - 1)){
237             while(current_row < (floor[0].length-1))
238                 floor[current_row++] [current_col] = true;
239             }
240         else while(spaces-- > 0)
241             floor[current_row++] [current_col] = true;
242         break;
243         case EAST:  if((current_col + spaces) >= (floor.length - 1)){
244             while(current_col < (floor.length-1))
245                 floor[current_row] [current_col++] = true;
246             }
247         else while(spaces-- > 0)
248             floor[current_row] [current_col++] = true;
249         break;
250         case WEST:  if((current_col - spaces) <= 0){
251             while(current_col >= 1)
252                 floor[current_row] [current_col--] = true;
253             }
254         else while(spaces-- > 0)
255             floor[current_row] [current_col--] = true;
256         break;
257     }
258     break;
259 }
260 }
261
262 /**
263  * Prints the floor array pattern to the console
264  * @see #floor
265  */
266 private void printFloor(){
267     for(int i = 0; i<floor.length; i++){
268         for(int j=0; j<floor[i].length; j++){
269             if(floor[i][j] == true)
270                 System.out.print('*');
271             else System.out.print('0');
272         }
273         System.out.println();
274     }
275 }
276
277 /**
278  * Exits the RobotRat program
279  * @see System#exit
280  */
281 private void exit(){
282     System.exit(0);
283 }
284
285
286 /**
287  * Prints error text message when invalid RobotRat menu item entered
288  */
289 private void printErrorMessage(){
290     System.out.println("Please enter a valid menu choice!");
291 }
292
293
294 /**
295  * The main method. This makes RobotRat an application.
296  */
297 public static void main(String[] args){
298     RobotRat rr = new RobotRat(20, 20);
299     rr.run();
300 }
301 } //end RobotRat class definition

```

SUMMARY

Use the project-approach strategy to systematically produce a solution to a programming problem. The purpose of the project-approach strategy is to help novice programmers maintain a sense of forward momentum during their project-development activities. The project-approach strategy comprises four strategy areas: application requirements, problem domain, language features, and high-level design and implementation strategy. The project approach strategy can be tailored to suit your needs.

Use the development-cycle to methodically implement your projects. Apply the plan, code, test, integrate steps iteratively in a tight-spiral fashion.

When you've completed your project, review it to ensure it meets all submission requirements related to comment headers, formatting, neatness, and the way it appears when printed. If you submit your project on a disk or another type of removable media, double-check to ensure all the class files are included. Use javadoc comments to generate professional-grade documentation.

Skill-Building Exercises

1. **Project-Approach Strategy:** Review the four project-approach strategy areas. Write a brief explanation of the purpose of each strategy area.
2. **Project-Approach Strategy:** Tailor the project-approach strategy to better suit your needs. What strategy areas would you add, delete, or modify? Explain your rationale.
3. **Development Cycle:** Review the phases of the development cycle. Write a brief description of each phase.
4. **UML Tool:** Obtain a Unified Modeling Language tool. Explore its capabilities. Use it to create class and state transition diagrams.

SUGGESTED PROJECTS

1. **Project-Approach Strategy & Development Cycle:** Apply the project-approach strategy and development cycle to your programming projects.

SELF-TEST QUESTIONS

1. List and describe the four project-approach strategy areas.
2. What is the purpose of the project-approach strategy?
3. List and describe the four phases of the development cycle demonstrated in this chapter.
4. How should the development cycle be applied to a programming project?
5. What is method stubbing?
6. What is the purpose of method stubbing?

7. When should you first compile and test your programming project?
8. What is a javadoc comment?
9. List at least three aspects of your project you should double-check before your turn it in to your instructor.
10. What is the purpose of a state transition diagram?

REFERENCES

Sun, Inc. online reference for the Java 2 Software Development Kit [java.sun.com]

NOTES

CHAPTER 4



Chess Anyone?

COMPUTERS, PROGRAMS, & ALGORITHMS

LEARNING OBJECTIVES

- STATE THE PURPOSE AND USE OF A COMPUTER
- STATE THE PRIMARY CHARACTERISTIC THAT MAKES THE COMPUTER A UNIQUE DEVICE
- LIST AND DESCRIBE THE FOUR STAGES OF THE PROGRAM EXECUTION CYCLE
- EXPLAIN HOW A COMPUTER STORES AND RETRIEVES PROGRAMS FOR EXECUTION
- STATE THE DIFFERENCE BETWEEN A COMPUTER AND A COMPUTER SYSTEM
- DEFINE THE CONCEPT OF A PROGRAM FROM THE HUMAN AND COMPUTER PERSPECTIVE
- STATE THE PURPOSE AND USE OF MAIN, AUXILIARY, AND CACHE MEMORY
- DESCRIBE HOW PROGRAMS ARE LOADED INTO MAIN MEMORY AND EXECUTED BY A COMPUTER.
- STATE THE PURPOSE AND USE OF THE JAVA VIRTUAL MACHINE
- LIST THE SIMILARITIES BETWEEN THE JAVA VIRTUAL MACHINE AND A REAL COMPUTER
- DESCRIBE THE ARCHITECTURE OF THE JAVA HOTSPOT™ VIRTUAL MACHINE
- EXPLAIN THE PURPOSE OF BYTE CODE
- DEFINE THE CONCEPT OF AN ALGORITHM

INTRODUCTION

Computers, programs, and algorithms are three closely related topics that deserve special attention before you start learning about Java proper. Why? Simply put, computers execute programs, and programs implement algorithms. As a programmer, you will live your life in the world of computers, programs, and algorithms.

As you progress through your studies, you will find it extremely helpful to understand what makes a computer a computer, what particular feature makes a computer a truly remarkable device, and how one functions from a programmer's point of view. You will also find it helpful to know how humans view programs, and how human-readable program instructions are translated into a computer-executable form.

Next, it will be imperative for you to thoroughly understand the concept of an algorithm and to understand how good and bad algorithms ultimately affect program performance.

Finally, I will show you how Java programs are transformed into byte code and executed by a Java virtual machine. Armed with a fundamental understanding of computers, programs, and algorithms, you will be better prepared to understand the concepts of the Java virtual machine as well as its execution performance and security ramifications.

WHAT IS A COMPUTER?

A computer is a device whose function, purpose, and behavior is prescribed, controlled, or changed via a set of stored instructions. A computer can also be described as a general-purpose machine. One minute a computer may execute instructions making it function as a word processor or page-layout machine. The next minute it might be functioning as a digital canvas for an artist. Again, this functionality is implemented as a series of instructions. Indeed, in each case the only difference between the computer functioning as a word processor and the same computer functioning as a digital canvas is in the set of instructions the computer is executing.

COMPUTER VS. COMPUTER SYSTEM

Due to the ever-shrinking size of the modern computer it is often difficult for students to separate the concept of the computer from the computer system in which it resides. As a programmer, you will be concerned with both. You will need to understand issues related to the particular processor that powers a computer system in addition to issues related to the computer system as a whole. Luckily though, as a Java programmer, you can be extremely productive armed with only a high-level understanding of each. Ultimately, I highly recommend spending the time required to get intimately familiar with how your computer operates. For this chapter I will use the Apple Power Mac G4 as an example, but the concepts are the same for any computer or computer system.

COMPUTER SYSTEM

A typical Power Mac G4 computer system is pictured in figure 4-1.



Figure 4-1: Typical Power Mac G4 System

The computer system comprises the system unit, monitor, speakers, keyboard, and mouse. The computer system also includes any operating system or utility software required to make all the components work together.

The system unit houses the processor, power supply, internal hard disk drives, memory, and other system components required to interface the computer to the outside world. These interface components consume the majority of available space within the system unit as shown in figure 4-2.

Image courtesy Apple Computer, Inc.



Figure 4-2: System Unit

The processor is connected to the system unit's main logic board. Electronic pathways called buses connect the processor to the various interface components. Other miscellaneous electronic components are located on the main logic board to control the flow of communication between the processor and the outside world. Figure 4-3 is a block diagram of a Power Mac G4 main logic board.

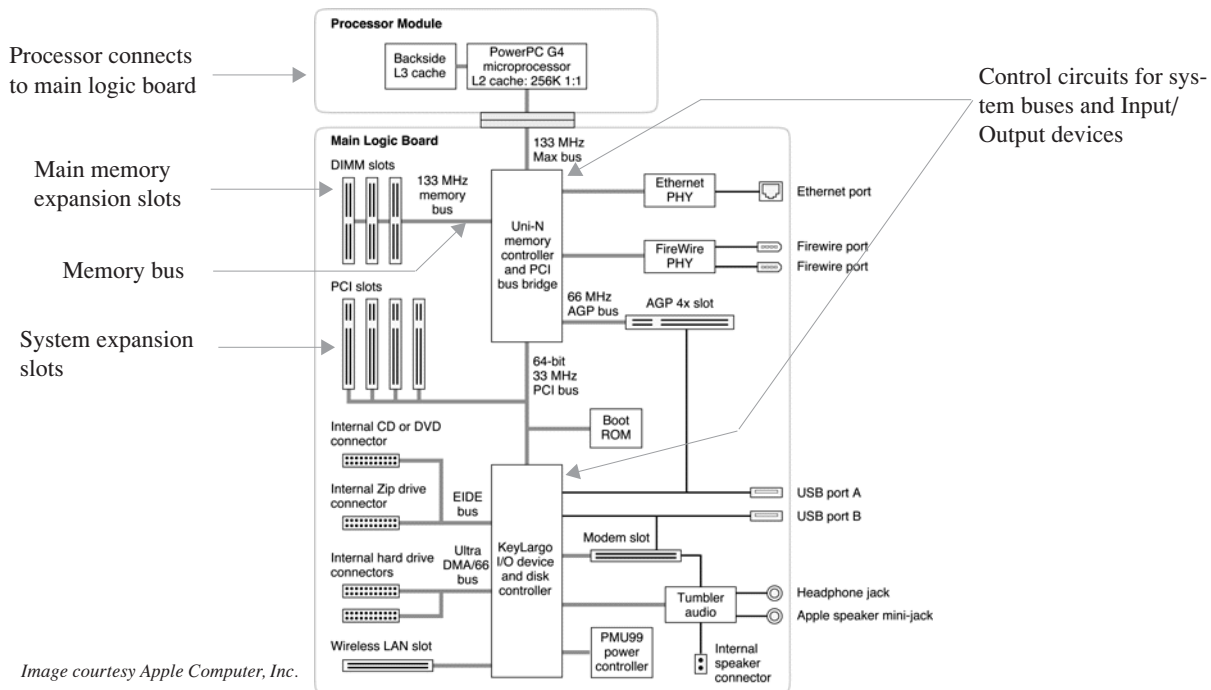


Image courtesy Apple Computer, Inc.

Figure 4-3: Main Logic Board Block Diagram

Figure 4-3 does a good job of highlighting the number of computer system support components required to help the processor do its job. The main logic board supports the addition of main memory, auxiliary storage devices, communication devices such as a modem, a wireless local area network card as well as a standard Ethernet port, keyboard, mouse, speakers, microphones, Firewire devices, and, with the insertion of the appropriate third party system expansion card, just about any other functionality you can imagine. All this system functionality is supported by the main logic board, but the heart of the system is the PowerPC G4 processor. Let's take a closer look.

PROCESSOR

If you lift up the heat sink pictured in figure 4-2 you would see a PowerPC G4 processor like the one shown in figure 4-4.

The PowerPC G4 7400 microprocessor pictured here runs at speeds between 350 and 500 megahertz with a Millions of Instructions per Second (MIPS) rating of 917 MIPS at 500 megahertz. The G4 is a powerful processor, yet at the time of this writing there are more powerful processors on the market, namely the G5!

The 7400 processor is a Reduced Instruction Set Computer (RISC) meaning its architecture is optimized to execute instructions in as few clock cycles as possible. The block diagram for the 7400 is shown in figure 4-5 and is even more impressive than that of the main logic board. The G4 is a superscalar, pipelined processor. It is superscalar in that it can pull two instructions from the incoming instruction stream and execute them simultaneously. It is pipelined in that instruction execution is broken down into discrete steps meaning several instructions can be in the pipeline at any given moment at different stages of execution.



Figure 4-4: PowerPC G4 Processor

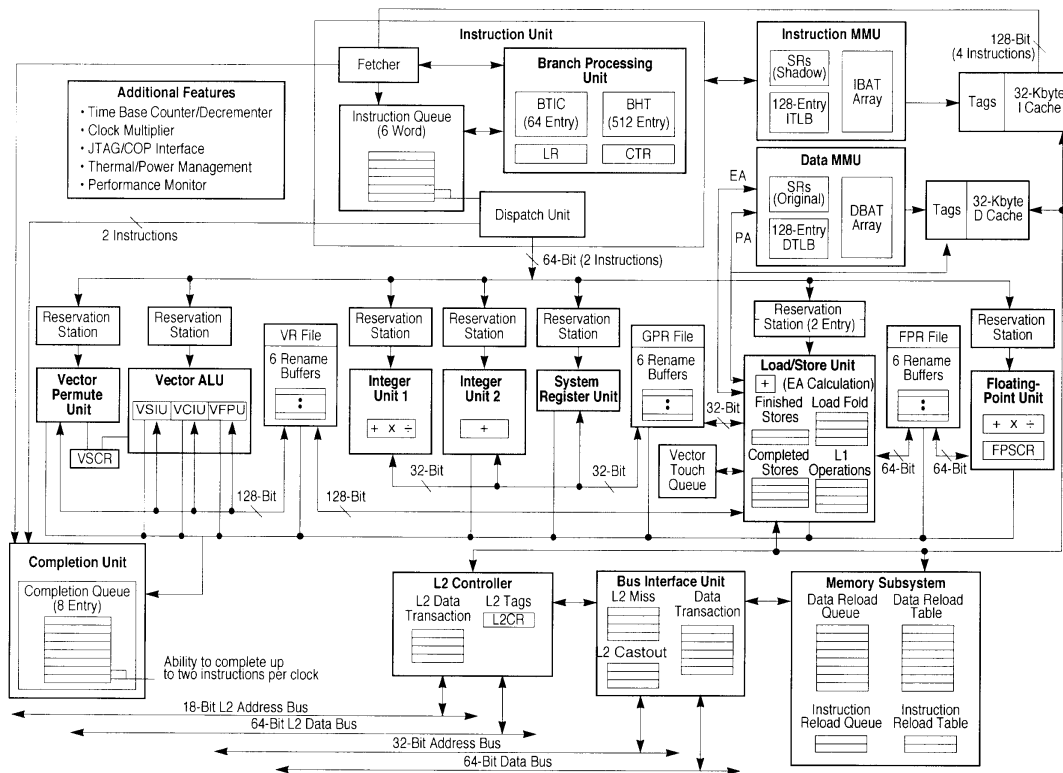


Figure 4-5: Motorola PowerPC 7400 Block Diagram

THREE ASPECTS OF PROCESSOR ARCHITECTURE

There are generally three aspects of processor architecture programmers should be aware of: feature set, feature set implementation, and feature set accessibility.

FEATURE SET

A processor's feature set is derived from its design. Can floating point arithmetic be executed in hardware or must it be emulated in software? Must all data pass through the processor or can input/output be handled off chip while the processor goes about its business? How much memory can the processor access? How fast can it run? How much data can it process per unit time? A processor's design addresses these and other feature set issues.

FEATURE SET IMPLEMENTATION

This aspect of computer architecture is concerned primarily with how processor functionality is arranged and executed in hardware. How does the processor implement the feature set? Is it a Reduced Instruction Set Computer (RISC), or Complex Instruction Set Computer (CISC)? Is it superscalar and pipelined? Does it have a vector execution unit? Is the floating-point unit on the chip with the processor or does it sit off to the side? Is the super-fast cache memory part of the processor or is it located on another chip? These questions all deal with how processor functionality is achieved or how its design is executed.

FEATURE SET ACCESSIBILITY

Feature set accessibility is the aspect of a processor's architecture you are most concerned with as a programmer. Processor designers make a processor's feature set available to programmers via the processor's instruction set. A valid instruction in a processor's raw instruction set is a set of voltage levels that, when decoded by the processor, have special meaning. A high voltage is usually translated as “on” or “1” and a low voltage is usually translated as “off” or “0”. A set of on and off voltages is conveniently represented to humans as a string of ones and zeros. Instructions in this format are generally referred to as machine instructions or machine code. However, as processor power increases, the size of machine instructions grow as well, making it extremely difficult for programmers to deal directly with machine code.

FROM MACHINE CODE TO ASSEMBLY

To make a processor's instruction set easier for humans to understand and work with each machine instruction is represented symbolically in a set of instructions referred to as an assembly language. To the programmer, assembly language represents an abstraction or a layer between programmer and machine intended to make the act of programming more efficient. Programs written in assembly language must be translated into machine instructions before being executed by the processor. A program called an assembler translates assembly language into machine code.

Although assembly language is easier to work with than machine code it requires a lot of effort to crank out a program in assembly code. Assembly language programmers must busy themselves with issues like register usage and stack conventions.

High-level programming languages like Java add yet another layer of abstraction. Java, with its object-oriented language features, lets programmers think in terms of solving the problem at hand, not in terms of the processor or the machine code that it's ultimately executing.

MEMORY ORGANIZATION

Modern computer systems have similar memory organizations and as a programmer you should be aware of how computer memory is organized and accessed. The best way to get a good feel for how your computer works is to poke around in memory and see what's in there for yourself. This section provides a brief introduction to computer memory concepts to help get you started.

MEMORY BASICS

A computer's memory stores information in the form of electronic voltages. There are two general types of memory: volatile and non-volatile. Volatile memory will lose any information stored there if power is removed for any length of time. Main memory and cache memory, two forms of Random Access Memory (RAM), are examples of volatile memory. Read Only Memory (ROM) and auxiliary storage devices such as CD ROMs, DVDs, hard disk drives, memory sticks, floppy disks, and tapes, are examples of non-volatile memory.

MEMORY HIERARCHY

Computer systems contain several different types of memory. These memory types range from slow and cheap to fast and expensive. The proportion of slow cheap memory to fast expensive memory can be viewed in the shape of a pyramid commonly referred to as the memory hierarchy as is shown in figure 4-6.

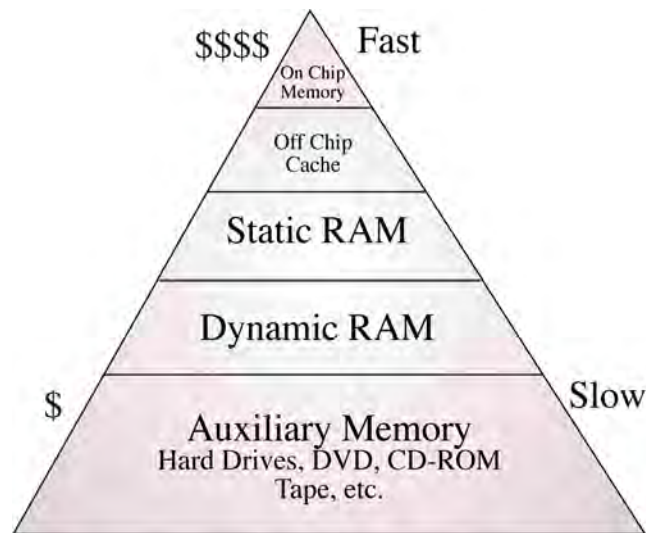


Figure 4-6: Memory Hierarchy

The job of a computer-system designer, with regards to memory subsystems, is to make the whole computer perform as if all the memory were fast and expensive. Thus they utilize cache memory to store frequently used data and instructions and buffer disk reads to memory to give the appearance of faster disk access. Figure 4-7 shows a block diagram of the different types of memory used in a typical computer system.

During program execution, the faster cache memory is checked first for any requested data or instruction. If it's not there, a performance penalty is extracted in the form of longer overall access times required to retrieve the information from a slower memory source.

Bits, Bytes, Words

Program code and data are stored in main memory as electronic voltages. Since I'm talking about digital computers the voltages levels represent two discrete states depending on the level. Usually low voltages represent no value, off, or 0, while a high voltage represents on, or 1.

When program code and data is stored on auxiliary memory devices, electronic voltages are translated into either electromagnetic fields (tape drives, floppy and hard disks) or bumps that can be detected by laser beam (CDs, DVDs, etc.).

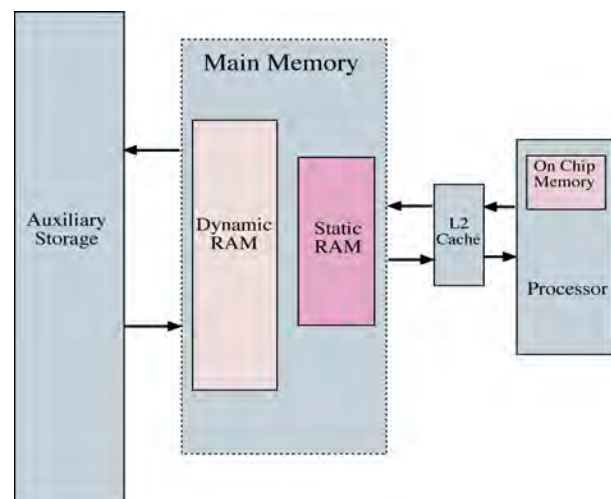


Figure 4-7: Simplified Memory Subsystem Diagram

Bit

The bit represents one discrete piece of information stored in a computer. On most modern computer systems bits cannot be individually accessed from memory. However, after the byte to which a bit belongs is loaded into the processor the byte can be manipulated to access a particular bit.

Byte

A byte contains 8 bits. Most computer memory is byte addressable, although as processors become increasingly powerful and can manipulate wider memory words, loading bytes by themselves into the processor becomes increasingly inefficient. This is the case with the G4 processor and for that reason the fastest memory reads can be done a word at a time.

Word

A word is a collection of bytes. The number of bytes that comprise a word is computer-system dependent. If a computer's data bus is 32 bits wide and its processor's registers are 32 bits wide, then the word size would be 4 bytes long. ($32 \text{ bits} / 8 \text{ bits} = 4 \text{ bytes}$) Bigger computers will have larger word sizes. This means they can manipulate more information per unit time than a computer with a smaller word size.

ALIGNMENT AND ADDRESSABILITY

You can expect to find your computer system's memory to be byte addressable and word aligned. Figure 4-8 shows a simplified diagram of a main memory divided into bytes and the different buses connecting it to the processor. In this diagram the word size is 32 bits wide.

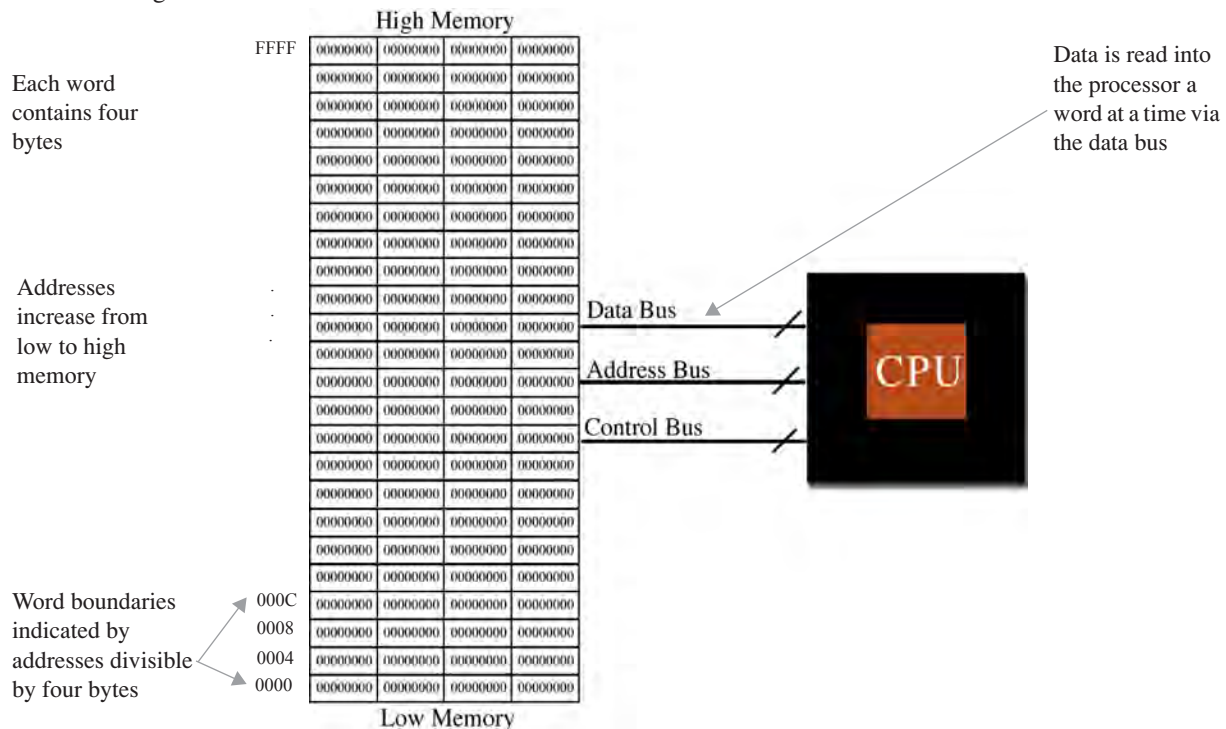


Figure 4-8: Simplified Main Memory Diagram

The memory is byte addressable in that each byte can be individually accessed although the entire word that contains the byte is read into the processor. Data in memory can be aligned for efficient manipulation. Alignment can be to natural or some other boundary. For example, on a PowerPC system, contents of memory assigned to instances of structures is aligned to natural boundaries meaning a one-byte data element will be aligned to a one-byte boundary. A two-byte element would be aligned to a two-byte boundary. Individual data elements not belonging to structures are

usually aligned to four-byte boundaries. (*Note: The term “structure” denotes a C or C++ structure. In Java there is no equivalent and the Java virtual machine prevents direct memory manipulation.*)

WHAT IS A PROGRAM?

Intuitively you already know the answer to this question. A program is something that runs on a computer. This simple definition works well enough for most purposes, but as a programmer you will need to arm yourself with a better understanding of exactly what makes a program a program. In this section I will discuss programs from two aspects: the computer and the human. You will find this information extremely helpful and it will tide you over until you take a formal course on computer architecture.

TWO VIEWS OF A PROGRAM

A program is a set of programming language instructions plus any data the instructions act upon or manipulate. This is a reasonable definition and if you are a human it will do. If you are a processor it will just not fly. That’s because humans are great abstract thinkers and computers are not, so it is helpful to view the definition of a program from two points of view.

THE HUMAN PERSPECTIVE

Humans are the masters of abstract thought; it is the hallmark of our intelligence. High-level, object-oriented languages like Java give us the ability to analyze a problem abstractly and symbolically express its solution in a form that is both understandable by humans and readable by other programs. By other programs I mean the Java code a programmer writes must be translated from Java into machine instructions recognizable by a particular processor. This translation is effected by running a compiler that converts the Java code into bytecode that is then executed by a Java virtual machine.

To a Java programmer a program is a collection of classes that model the behavior of objects in a particular problem domain. These classes model object behavior by defining object attributes (data) and class methods to manipulate the class attributes. On an even higher level, a program can be viewed as an interaction between objects. This view of a program is convenient for humans.

THE COMPUTER PERSPECTIVE

From the computer’s perspective a program is simply machine instructions and data. Usually both the instructions and data reside in the same memory space. This is referred to as a Von Neumann architecture. In order for a program to run it must be loaded into main memory and the address of the first instruction of the program given to the processor. In the early days of computing, programs were coded into computers by hand and then executed. Nowadays all the nasty details of loading programs from auxiliary memory into main memory are handled by an operating system, which, by the way, is a program.

Since both instructions and data reside in main memory how does a computer know when it is dealing with an instruction or with data? The answer to this question will be discussed in detail below but here’s a quick answer: It depends on what the computer is expecting. If a computer reads a memory location expecting to find an instruction and it does, everything runs fine. The instruction is decoded and executed. If it reads a memory location expecting to find an instruction but instead finds garbage, then the decode fails and the computer might lock up!

THE PROCESSING CYCLE

Computers are powerful because they can do repetitive things really fast. When the executable code is loaded into main memory the processor can start executing the machine instructions. When a computer executes or runs a program it constantly repeats a series of processing steps commonly referred to as the processing cycle. The processing cycle consists of four primary steps: instruction fetch, instruction decode, instruction execution, and result store.

The step names can be shortened to simply fetch, decode, execute, and store. Different processors will implement the processing cycle differently but for the most part these four processing steps are being carried out in some form or another. The processing cycle is depicted in figure 4-9.

FETCH

In the fetch step, an instruction is read from memory and presented to the processor's decode section. If cache memory is present it is checked first. If the requested memory address contents resides in the cache the read operation will execute quickly. Otherwise, the processor will have to wait for the data to be accessed from the slower main memory.

DECODE

In the decode step, the instruction fetched from memory is decoded. If the computer thinks it is getting an instruction but instead it gets garbage there will be problems. A computer system's ability to recover from such situations is generally a function of a robust operating system.

EXECUTE

If the fetched instruction is successfully decoded, meaning it is a valid instruction in the processor's instruction set, it is executed. A computer is a bunch of electronic switches. Executing an instruction means the computer's electronic switches are turned either on or off to carry out the actions required by a particular instruction. (*Different instructions cause different sets of switches to be turned on or off.*)

STORE

After an instruction is executed the results of the execution, if any, must be stored somewhere. Most arithmetic instructions leave the result in one of the processor's onboard registers. Memory write instructions would then be used to transfer the results to main memory. Keep in mind that there is only so much storage space inside a processor. At any given time, almost all data and instructions reside in main memory, and are only loaded into the processor when needed.

Why A PROGRAM CRASHES

Notwithstanding catastrophic hardware failure, a computer crashes or locks up because what it was told was an instruction was not! The faulty instruction loaded from memory turns out to be an unrecognizable string of ones and zeros and when it fails to decode into a proper instruction the computer halts.

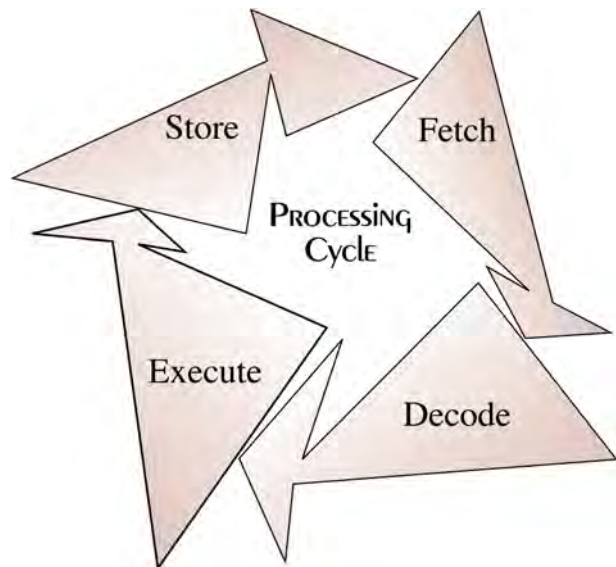


Figure 4-9: Processing Cycle

ALGORITHMS

Computers run programs; programs implement algorithms. A good working definition of an algorithm for the purpose of this book is that an algorithm is a recipe for getting something done on a computer. Pretty much every line of source code you write is considered part of an algorithm. What I'd like to do in this brief section is bring to your attention the concept of good vs. bad algorithms.

Good vs. Bad Algorithms

There are good ways to do something in source code and there are bad ways to do the same exact thing. A good example of this can be found in the act of sorting. Suppose you want to sort in ascending order the following list of integers:

1, 10, 7, 3, 9, 2, 4, 6, 5, 8, 0, 11

One algorithm for doing the sort might go something like this:

Step 1: Select the first integer position in the list

Step 2: Compare the selected integer with its immediate neighbor

Step 2.2: If the selected integer is greater than its neighbor swap the two integers

Step 2.3: Else, leave it where it is

Step 3: Continue comparing selected integer position with all other integers repeating steps 2.2 - 2.3

Step 4: Select the second integer position on the list and repeat the procedure beginning at step 2

Continue in this fashion until all integers have been compared to all other integers in the list and have been placed in their proper position.

This algorithm is simple and straightforward. It also runs pretty fast for small lists of integers but it is really slow given large lists of integers to sort. Another sorting algorithm to sort the same list of integers goes as follows:

Step 1: Split the list into two equal sublists

Step 2: Repeat step 1 if any sublist contains more than two integers

Step 3: Sort each sublist of two integers

Step 4: Combine sorted sublists until all sorted sublists have been combined

This algorithm runs a little slow on small lists because of all the list splitting going on but sorts large lists of integers way faster than the first algorithm. The first algorithm lists the steps for a routine I call dumb sort. Example 4.1 gives the source code for a short program that implements the dumb sort algorithm.

4.1 DumbSort.java

```

1      public class DumbSort{
2          public static void main(String[] args){
3
4              int a[] = {1,10,7,3,9,2,4,6,5,8,0,11};
5
6              int innerloop = 0;
7              int outerloop = 0;
8              int swaps = 0;
9
10             for(int i = 0; i<12; i++){
11                 outerloop++;
12                 for(int j = 1; j<12; j++){
13                     innerloop++;
14                     if(a[j-1] > a[j]) {
15                         int temp = a[j-1];
16                         a[j-1] = a[j];
17                         a[j] = temp;
18                         swaps++;}}
19
20             for(int i = 0; i<12; i++)
21                 System.out.print(a[i] + " ");
22
23             System.out.println();
24             System.out.println("Outer loop executed " + outerloop + " times.");
25             System.out.println("Inner loop executed " + innerloop + " times.");
26             System.out.println(swaps + " swaps completed.");
27         }
28     }

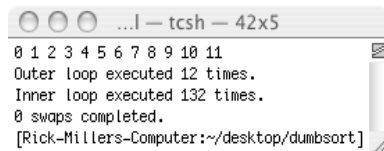
```

Included in the dumb sort test source code are a few variables intended to help collect statistics during execution. These are `innerloop`, `outerloop`, and `swaps` declared on lines 6, 7, and 8 respectively. Figure 4-10 gives the results from running the dumb sort test program.

Notice that the inner loop executed 132 times and that 30 swaps were conducted. Can the algorithm run any better? One way to check is to rearrange the order of the integers in the array. What if the list of integers is already sorted? Figure 4-11 gives the results of running dumb sort on an already sorted list of integers:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

It appears that both the outer loop and inner loop are executed the same number of times in each case, which is of course the way the source code is written, but it did run a little faster because fewer swaps were necessary.



```

0 1 2 3 4 5 6 7 8 9 10 11
Outer loop executed 12 times.
Inner loop executed 132 times.
0 swaps completed.
[Rick-Millers-Computer:~/desktop/dumbsort]

```

Figure 4-11: Dumb Sort Results 2

Can the algorithm run any worse? What if the list of integers is completely unsorted? Figure 4-12 gives the results of running dumb sort on a completely unsorted list:

11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

The outer loop and inner loop executed the same number of times but 66 swaps were necessary to put everything in ascending order. So it did run a little slower this time.

In dumb sort, because we're sorting a list of 12 integers, the inner loop executes 12 times for every time the outer loop executes. If dumb sort needed to sort 10,000 integers then the inner loop would need to execute 10,000 times for every time the outer loop executed. To generalize the performance of dumb sort you could say that for some number N integers to sort, dumb sort executes the inner loop roughly $N \times N$ times. There is some other stuff going on besides loop iterations but when N gets really large, the loop iteration becomes the overwhelming measure of dumb sort's performance as a sorting algorithm. Computer scientists would say that dumb sort has order N^2 performance. Saying it another way, for a really large list of integers to sort, the time it takes dumb sort to do its job is approximately the square of the number N of integers that need to be sorted.

When an algorithm's running time is a function of the size of its input the term used to describe the growth in time to perform its job vs. the size of the input is called the growth rate. Figure 4-13 shows a plot of algorithms with the following growth rates: $\log n$, n , $n \log n$, n^2 , n^3 , n^n

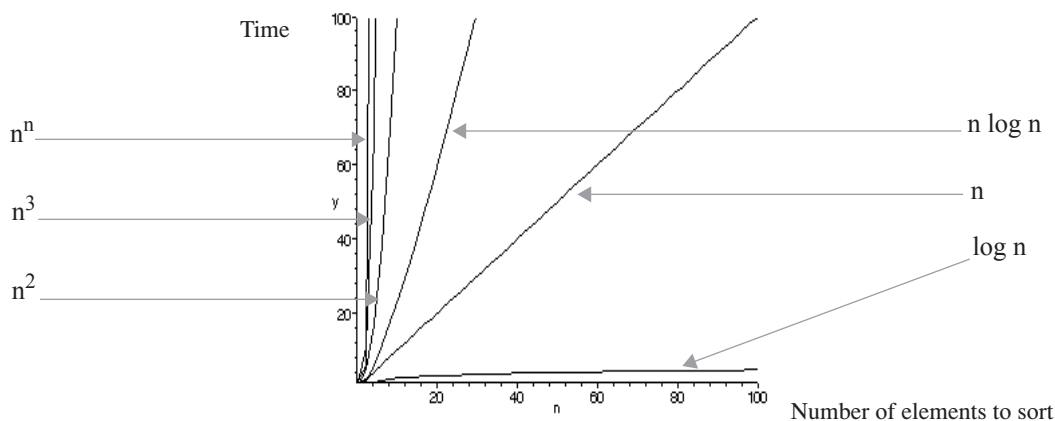


Figure 4-13: Algorithmic Growth Rates

As you can see from the graph, dumb sort, with a growth rate of n^2 , is a bad algorithm, but not as bad as some other algorithms. The good thing about dumb sort is that no matter how big its input grows, it will eventually sort all the integers. Sorting problems are easily solved. There are some problems, however, that defy straightforward algorithmic solution.

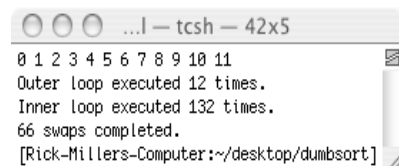


```

0 1 2 3 4 5 6 7 8 9 10 11
Outer loop executed 12 times.
Inner loop executed 132 times.
30 swaps completed.
[Rick-Millers-Computer:~/desktop/dumbsort]

```

Figure 4-10: Dumb Sort Results 1



```

0 1 2 3 4 5 6 7 8 9 10 11
Outer loop executed 12 times.
Inner loop executed 132 times.
66 swaps completed.
[Rick-Millers-Computer:~/desktop/dumbsort]

```

Figure 4-12: Dumb Sort Results 3

DON'T REINVENT THE WHEEL!

If you are new to programming the best advice I can offer is for you to seek the knowledge of those who have come before you. There are many good books on algorithms, some of which are listed in the reference section. Studying good algorithms helps you write better code.

THE JAVA HOTSPOT™ VIRTUAL MACHINE

The Java HotSpot™ virtual machine (JVM) is a program that runs Java programs. The JVM is targeted to work on specific hardware platforms and host operating systems such as Intel™ processors running the Microsoft Windows™ operating system or the PowerPC™ processor running Mac OS X as is shown in figure 4-14.

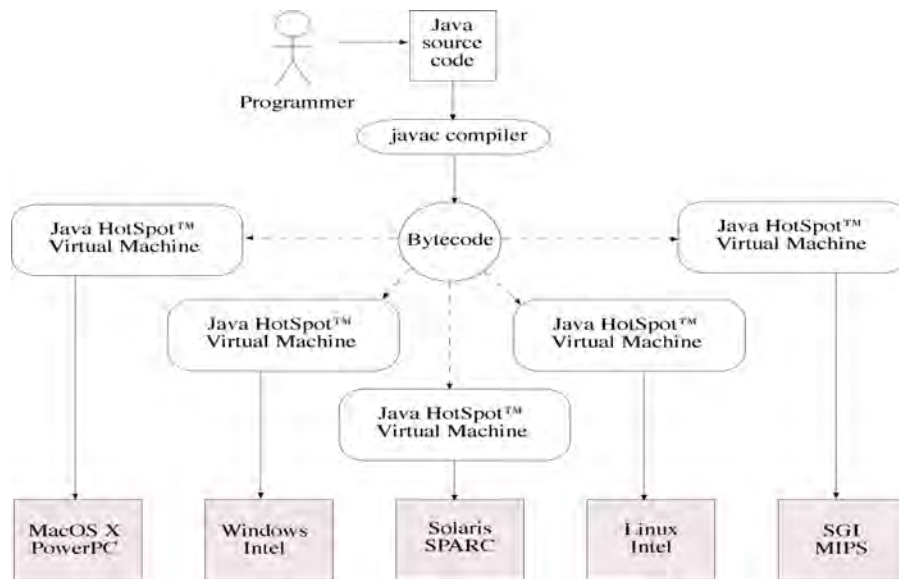


Figure 4-14: Java HotSpot™ Virtual Machine Targets Specific Hardware Platforms

As figure 4-14 illustrates, the concept of the virtual machine facilitates the write-once, run-anywhere nature of Java code. Programmers write Java programs that can run on any computing platform that implements a Java virtual machine.

OBTAINING THE JAVA HOTSPOT™ VIRTUAL MACHINE

When you download the Java Software Developers Kit (SDK) it comes with a Java Runtime Environment (JRE) that includes two versions of the JVM: a client version and a server version.

CLIENT & SERVER VIRTUAL MACHINES

When you run Java programs with the java command-line tool you are executing them with the client version of the JVM, which is the default virtual machine. If you specify `-server` as an argument to the java command-line tool you will execute your programs using the server version of the JVM.

The difference between the two version of the JVM lies in how each version optimizes the java bytecode execution. The client version optimizes execution primarily to support fast program startups and graphical user interfaces (GUIs). The server optimizes execution to support often-repeated code snippets.

Classic VM vs. JIT vs. HotSpot™

The first several releases of the Java runtime environment shipped with what is now referred to as the classic virtual machine. The classic virtual machine ran Java bytecode in an interpreted mode, meaning each bytecode instruction was executed by the virtual machine. The virtual machine was responsible for translating each bytecode instruction into a series of virtual machine instructions that then performed the operation indicated by the bytecode instruction.

Later releases of the Java runtime environment added the services of a just-in-time (JIT) compiler to the virtual machine architecture. The purpose of the JIT is to compile bytecode instructions into processor native code that can be executed directly on the target hardware platform. The JIT provided improvements in execution speed but extracted an upfront penalty due to the compilation process since all bytecodes are compiled prior to execution.

The HotSpot virtual machine provides performance improvements over both the classic VM and the JIT. The HotSpot virtual machine utilizes the services of a bytecode profiler. As bytecode is executed by the virtual machine, an analysis is performed to determine the locations of critical bytecode sequences, or hotspots, and to then compile those bytecode hotspots into native code. Programs startup faster and, as execution progresses, also benefit from native processor execution speeds.

JAVA HOTSPOT™ VIRTUAL MACHINE ARCHITECTURE

Figure 4-15 shows a high-level Java HotSpot virtual machine architectural diagram. Bytecode loaded into and executed by the HotSpot VM is initially interpreted. As execution progresses information about the code's execution behavior is obtained by the profiler. Frequently executed bytecode sequences are compiled into host-computer native code. Bytecode evaluation continues throughout the execution lifetime of the program.

There are two versions of the adaptive compiler: one tuned specifically to compile client programs and one tuned to compile server programs. Use of the client compiler results in smaller program memory footprints and quicker program startup times. Use of the server compiler results in more stringent code optimizations to extract every ounce of performance.

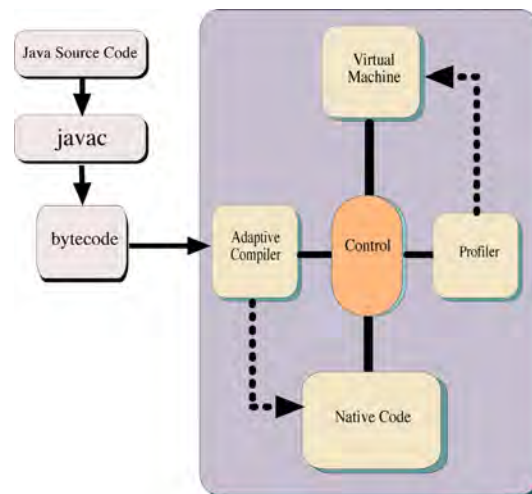


Figure 4-15: Java HotSpot™ Virtual Machine Architecture

SERVICES PROVIDED BY THE JAVA HOTSPOT™ VIRTUAL MACHINE AND THE JRE

The HotSpot virtual machine together with the JRE provides several important services to Java programs. These services include class loading, security, memory management and garbage collection, preemptive multithreading and thread synchronization, native method linking, and error and exception management.

SUMMARY

Computers run programs; programs implement algorithms. As a programmer you need to be aware of development issues regarding your computer system and the processor it is based on.

A computer system contains a processor, I/O devices, and supporting operating system software. The processor is the heart of the computer system.

Programs can be viewed from two perspectives: human and computer. From the human perspective, programs, are a high-level solution statement to a particular problem. Object-oriented languages like Java help humans model extremely complex problems algorithmically. Java programs can also be viewed as the interaction between objects in a problem domain.

To a computer, programs are a sequence of machine instructions and data located in main memory. Processors run programs by rapidly executing the processing cycle of fetch, decode, execute, and store. If a processor expects an instruction and gets garbage it is likely to lock up. Robust operating systems can mitigate this problem to a certain degree.

There are bad algorithms and good algorithms. Study from the pros and you will improve your code writing skills.

The Java HotSpot™ virtual machine is a program that runs Java programs. Java HotSpot virtual machines are targeted for specific hardware platforms. There are two versions of the HotSpot virtual machine: one that utilizes the client version of the adaptive compiler and one that utilizes the server version of the adaptive compiler. The differences between the client and server compilers lie in the types of optimizations performed on the bytecode sequences.

Skill-Building Exercises

1. **Research Sorting Algorithms:** The second sorting algorithm listed on page 86 gives the steps for a merge sort. Obtain a book on algorithms, look for some Java code that implements the merge sort algorithm, and compare it to Dumb Sort. What's the growth rate for a merge sort algorithm? How does it compare to Dumb Sort's growth rate?
2. **Research Sorting Algorithms:** Look for an example of a bubble sort algorithm. How does the bubble sort algorithm compare to Dumb Sort? What small changes can be made to Dumb Sort to improve its performance to that of bubble sort? What percentage of improvement is obtained by making the code changes? Will it make a difference for large lists of integers?
3. **Research The Java HotSpot Virtual Machine:** Visit Sun's Java website [java.sun.com] and research the HotSpot virtual machine architecture. Become familiar with all the services the HotSpot virtual machine provides.

SUGGESTED PROJECTS

1. **Research Computer System:** Research your computer system. List all its components including the type of processor. Go to the processor manufacturer's web site and download developer information for your systems processor. Look for a block diagram of the processor and determine how many registers it has and their sizes. How does it get instructions and data from memory? How does it decode the instructions and process data?
2. **Compare Different Processors:** Select two different microprocessors and compare them to each other. List the feature set of each and how the architecture of each implements the feature set.

SELF-TEST QUESTIONS

1. List at least five components of a typical computer system.
2. What device do the peripheral components of a computer system exist to support?
3. From what two perspectives can programs be viewed? How does each perspective differ from the other?
4. List and describe the four steps of the processing cycle?
5. State in your own words the definition of an algorithm.
6. How does a processor's architecture serve to implement its feature set?
7. How can programmers access a processor's feature set?
8. State the purpose of the Java HotSpot virtual machine.
9. What is the difference between the client and server versions of the HotSpot virtual machine compilers?
10. How is the HotSpot virtual machine different from the Java Classic VM and the Just-in-Time (JIT) compiler?

REFERENCES

Motorola, Inc. *PowerPC 601 RISC Microprocessor User's Manual*

Motorola, Inc. *PowerPC 7400 RISC Microprocessor User's Manual*

Sedgewick, Robert. *Algorithms in C++*, Addison-Wesley Publishing Company, Reading Massachusetts, 1992, ISBN 0-201-51059-6.

Corman, Thomas, et. al. *Introduction To Algorithms*, The MIT Press, Cambridge, Massachusetts, 1990, ISBN 0-262-03141-8

Jon Meyer, et. al. *Java Virtual Machine*, O'Reilly & Associates, Inc. 1997. ISBN: 1-56592-194-1

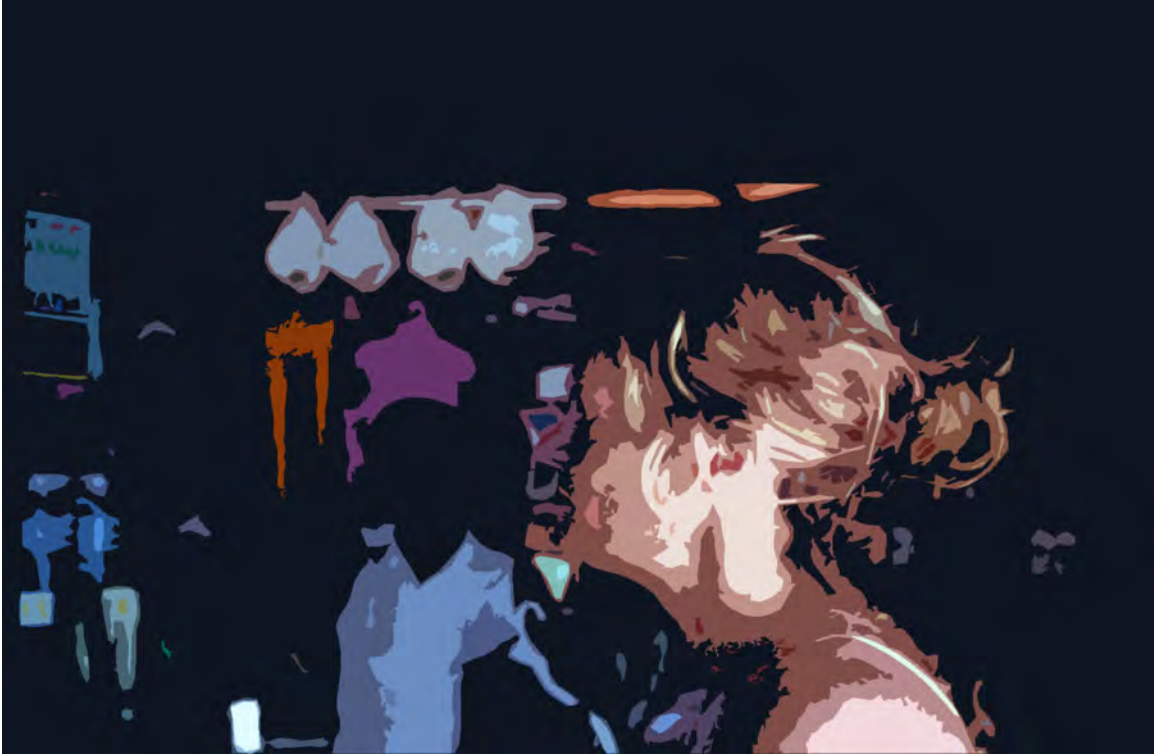
Java 2 SDK Standard Documentation: <http://java.sun.com/j2se/1.4.2/docs/index.html>

The Java HotSpot Virtual Machine v1.4.1, Technical White Paper, September 2002

NOTES

PART II: LANGUAGE FUNDAMENTALS

CHAPTER 5



Coralie In Shoppe

OVERVIEW OF THE JAVA PLATFORM API

LEARNING OBJECTIVES

- *List and describe the functionality provided by the primary packages of the Java 2 Standard Edition Platform Application Programming Interface (API)*
- *Describe how to trace a component's inheritance hierarchy to discover superclass functionality*
- *Describe how to obtain detailed information about the Java Platform API from the java.sun.com website*
- *State the definition of a deprecated method*

INTRODUCTION

The power of Java lies in the pre-existing functionality supplied by the Java platform Application Programming Interface (API). The Java platform API is a collection of packages that supply a set of classes ready for use in your programs. When you download and install the Java SDK or JRE you are also installing the Java platform API.

The Java platform is both a blessing and a curse to novice Java programmers. It's a blessing because with the classes supplied by the Java platform you can write extremely powerful programs with minimal effort. It's a curse because you must expend a lot of effort to learn your way around the Java platform API packages and the functionality they offer. In fact, of the total time you'll spend learning Java, you will spend about 10 percent of your time learning the syntax of the Java programming language, another 10 percent learning object-oriented programming concepts, and the other 80 percent of your time learning how to use the Java platform classes in your programs. But don't despair! You only need to know a small fraction of the Java platform API to write robust, well-behaved Java programs. Also, the Java platform classes you will use in this book will be introduced to you slowly as you progress through the text.

The purpose of this chapter then is to jump start your understanding of the Java platform API. I will show you how it is organized and what functionality it provides. I will also show you how to read a class inheritance hierarchy diagram so you can determine the full range of functionality a particular class provides. Knowing how to read a class inheritance diagram is a valuable skill that will pay huge dividends as you progress with your Java studies.

As I said above, in this book you will use just a small part of the Java Platform. However, once you know how to navigate the platform packages and how to read a class inheritance hierarchy you will have all the knowledge required to confidently approach the art of Java programming and add cool functionality to your programs.

JAVA PLATFORM API PACKAGES

Figure 5-1 offers an architectural overview of the Java platform version 1.4.2 packages and components.

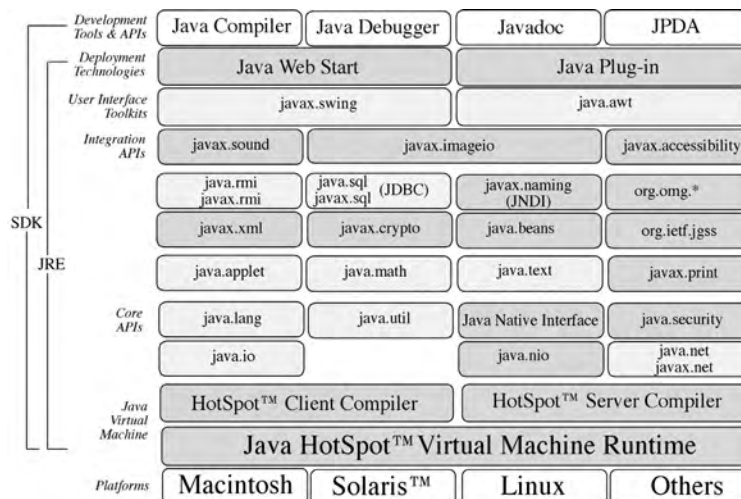


Figure 5-1: Java Platform Version 1.4.2 Package Architectural Overview

Notice that the difference between the Software Development Kit (SDK) and the Java Runtime Environment (JRE) is the presence of the additional development tools provided by the SDK. You will use the development tools provided by the SDK in conjunction with the support provided by the JRE to create and run your Java programs.

In addition to the Java HotSpot Virtual Machine, the client and server VM compilers, and the development tools of the SDK, the packages shown in light-grey and the classes they contain will be used frequently throughout this book. These include *javax.swing*, *java.awt*, *java.applet*, *java.rmi*, *java.sql (JDBC)*, *java.math*, *java.text*, *java.lang*, *java.util*, *java.io*, and *java.net*

The Java Native Interface (JNI) isn't a package per se; rather, it's a mechanism Java provides that enables you to write methods in programming languages like C or C++ and call them from your Java programs.

OBTAINING DETAILED JAVA PLATFORM API INFORMATION

Your best source for detailed information regarding all the packages of the Java Platform API is Sun's Java website: [java.sun.com] Figure 5-2 shows a screen shot from the Java 2 API specification page.

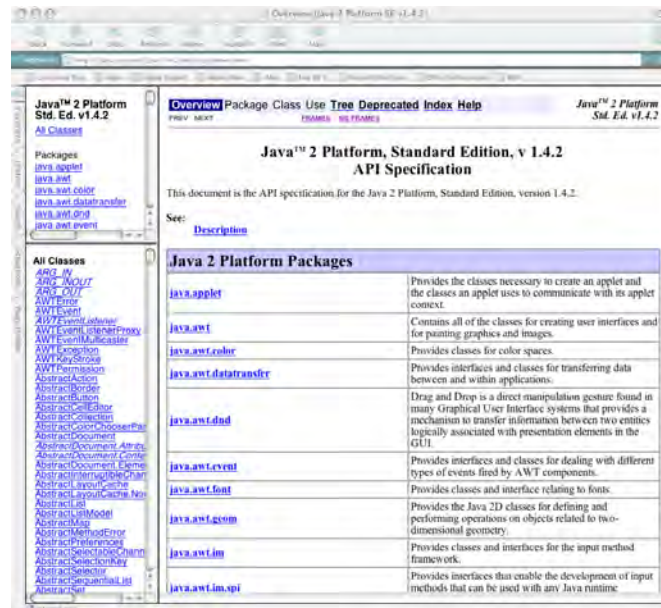


Figure 5-2: Java 2 Platform API Version 1.4.2 Specification Page

To obtain detailed information about the classes in a particular package simply click the package name link. Figure 5-3 shows a partial listing of the interfaces and classes of the `java.lang` package.

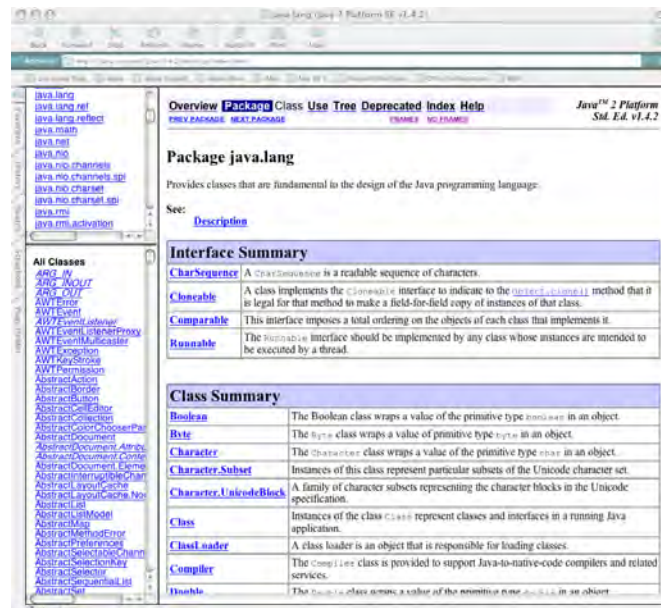


Figure 5-3: Partial Listing For `java.lang` Package

To drill down for more detailed information relating to a specific interface or class in a package click the class link. Figure 5-4 shows detailed information for the `java.lang.String` class.

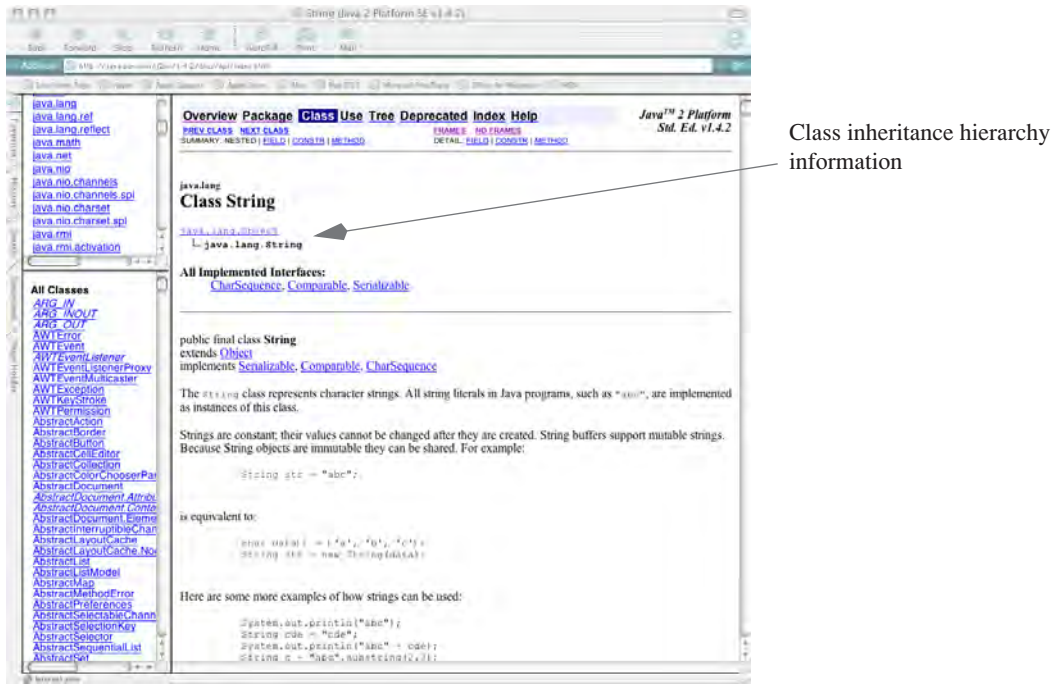


Figure 5-4: Detailed Information For java.lang.String class

The class detail page provides valuable information about inheritance hierarchy, usage examples, and constructor, method, and attribute indexes.

NAVIGATING A CLASS INHERITANCE HIERARCHY

Let’s dwell for a moment on the inheritance hierarchy information and how it will help you to know how to read, navigate, and interpret the inheritance hierarchy of a particular class.

All classes supplied by the Java platform inherit their functionality from one or more classes contained within the Java platform. The only exception to this rule is the `java.lang.Object` class which serves as the ultimate base class for all reference types.

In Java, classes can inherit the functionality of only one class, but may implement as many interfaces as necessary. The class it inherits from may also inherit its functionality from another class, and so on. What this means to you is that a class’s total functionality is the sum of all the public and protected methods it inherited from its base class plus any additional functionality it provides. Thus, to discover all the functionality provided by a particular class you must navigate up the inheritance hierarchy chain, visiting each class of the hierarchy in turn.

Let’s look at two examples. The first one is the `java.lang.String` class shown previously in figure 5-4. It inherits from the `java.lang.Object` class. So, the functionality provided by the `String` class is whatever functionality the `Object` class provides plus whatever functionality the `String` class adds or overrides. The `String` class also implements three interfaces: `CharSequence`, `Comparable`, and `Serializable`.

The inheritance hierarchy of the `String` class is shown in figure 5-5 as a Unified Modeling Language (UML) class diagram. By reading this class diagram you can immediately determine that a `String` is an `Object` and a `CharSequence`. It is also `Comparable` and `Serializable`. What this means from an object-oriented point of view is that anywhere in a Java program you can use an object of type `Object`, `CharSequence`, `Comparable`, or `Serializable` you can substitute a `String` object because it implements the functionality of all these types.

The second example is a little more complex but easy to understand now that you know how the `String` inheritance hierarchy works.

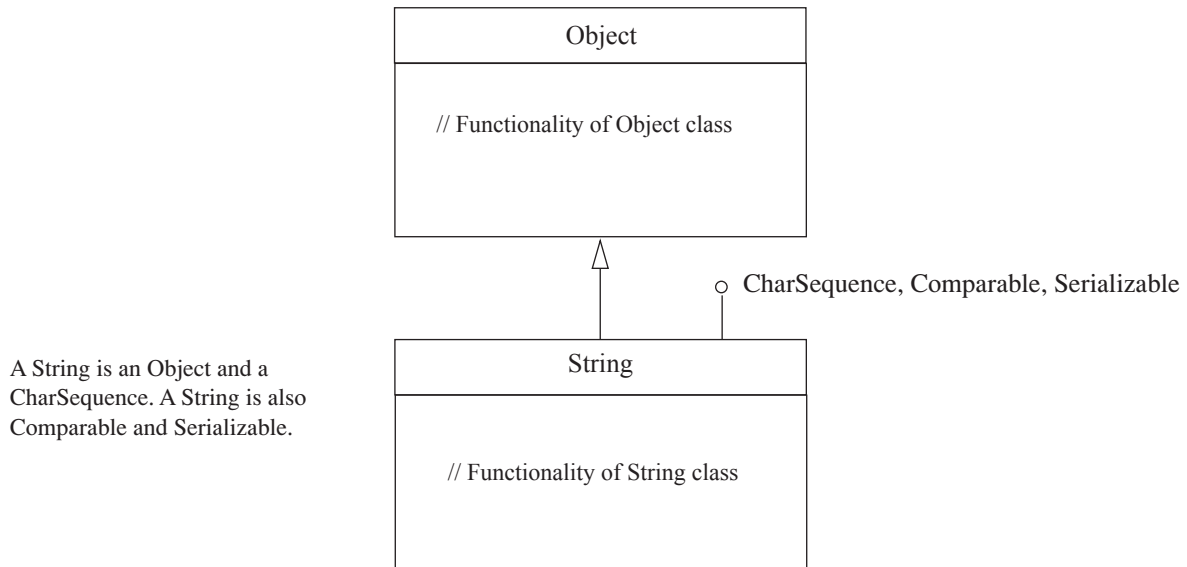


Figure 5-5: String Inheritance Hierarchy UML Diagram



Figure 5-6: JButton Inheritance Hierarchy Information

Some of the more complex inheritance hierarchies are found in the Java Foundation Class packages of `java.awt` and `javax.swing`. The classes in these packages are used to create rich graphical user interfaces (GUIs). Let's take a look at the `javax.swing.JButton` class inheritance hierarchy as is shown in figure 5-6

As you can tell from reading the inheritance hierarchy information provided in figure 5-6 the `JButton` class inherits its functionality from `javax.swing.AbstractButton`, which inherits from `javax.swing.JComponent`, which inherits from `java.awt.Container`, which in turn inherits from `java.awt.Component`, which ultimately inherits from `java.lang.Object`. It also implements the `Accessible`, `ImageObserver`, `ItemSelectable`, `MenuContainer`, `Serializable`, and `SwingConstants` interfaces. In other words, a `JButton` has a lot of functionality. To understand what a `JButton` can do you must understand what functionality is provided by all the base classes in the inheritance chain. You must also understand what it means to implement the various interfaces. All this information is available in the Java platform API documentation.

Figure 5-7 shows the UML class diagram for the `JButton` inheritance hierarchy. When you want to use a `JButton` in your program you will usually start by reading the `JButton` documentation to learn how to create `JButtons` and to learn what operations can be performed on `JButton` objects. However, to discover all that a `JButton` can do you must read the documentation for its direct base class `AbstractButton`, and in turn read the documentation for `JComponent`, `Container`, `Component`, and `Object` as well. Fortunately, the Java platform documentation takes steps to make this

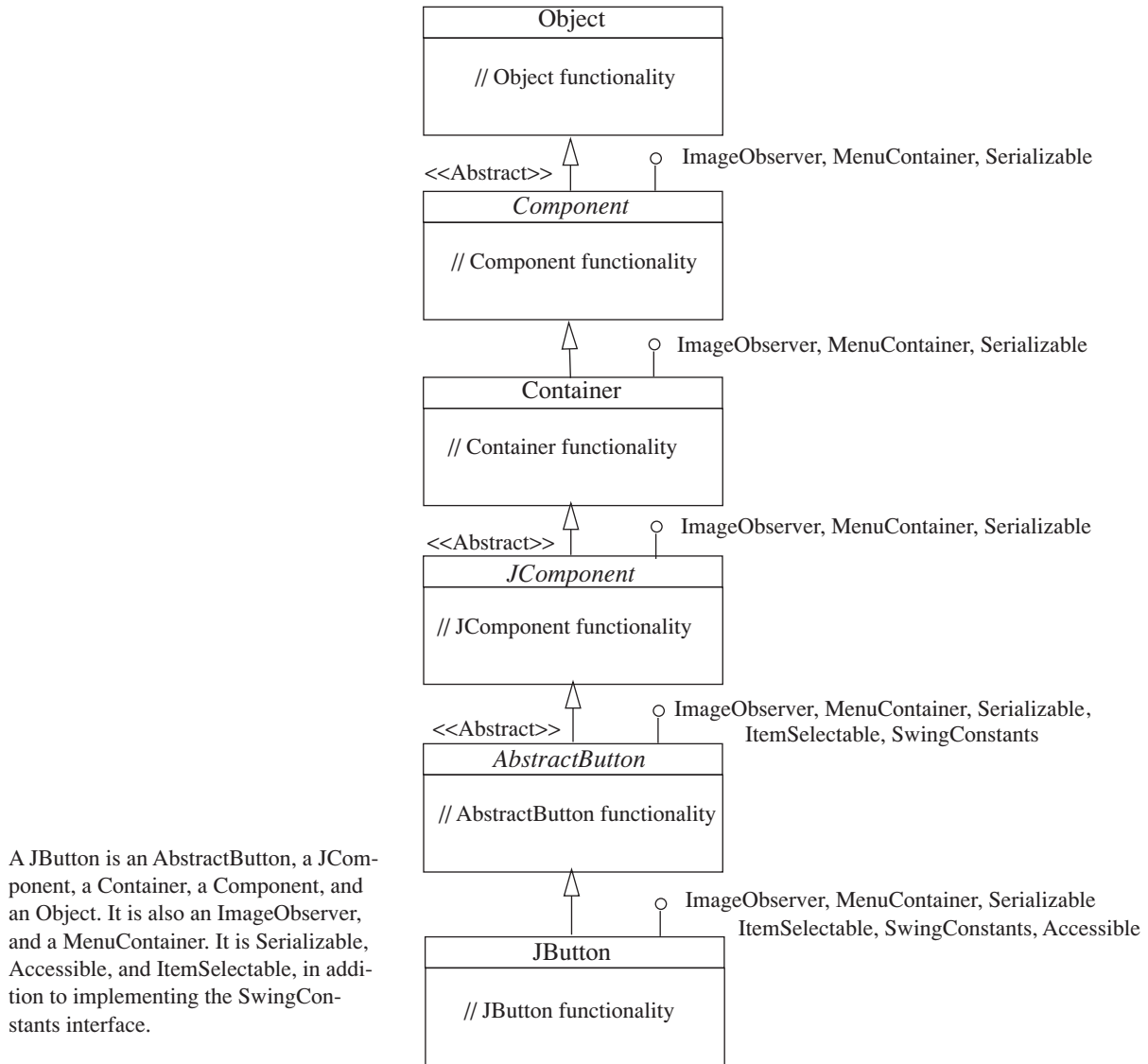


Figure 5-7: JButton Inheritance Hierarchy

process relatively painless by providing a section in each class detail page that lists the class’s inherited methods. Figure 5-8 shows a partial listing of the inherited methods for the JButton class.

The inherited methods section is nice because you can quickly scan the list of inherited methods to see if the functionality you require is already provided for you. For example, suppose you wanted to hide a JButton. You would look first for a hide() method in the JButton class and, guess what? — it’s not there. The next step would be to examine the inherited methods section of the JButton class documentation where you would find both the hide() and the show() methods in the java.awt.Component class. You click on the hide() method link and it takes you to the hide() method documentation of the Component class where you learn that the hide() method is deprecated as of Java version 1.1 and that you should use the setVisible() method instead. The setVisible() method description is shown in figure 5-9.

You will repeat this process over and over when you write Java programs. At first you will be driven to the brink of insanity by the need to look up every class and its associated methods, but, with practice, you will begin to remember what classes provide what functionality and in what package those classes are located.

Java platform. Although you can use a deprecated method in your code with no immediate ill effects, you're best advised not to, especially if you are writing production code.

JAVA PLATFORM PACKAGES USED IN THIS BOOK

This section provides a quick overview of the Java platform API packages you will be exposed to as you progress through this book. The packages and some of their important functionality are listed in table 5-1.

Package	Functionality
java.applet	Provides the classes required to write Java applets. It contains the Applet class that serves as the base class for the javax.swing.JApplet class.
java.awt	The Abstract Windowing Toolkit (AWT) provides the classes necessary to create user interfaces. The javax.swing package uses several AWT components as base classes for swing components.
java.rmi	Provides classes required to write Remote Method Invocation (RMI) programs.
java.awt.event	Provides classes and interfaces required for handling events fired by AWT and Swing components. You will use the classes and interfaces in the java.awt.event package to write event-driven programs.
java.awt.image	Provides classes required to create and manipulate images.
java.sql	Provides classes required to write programs that use Java Database Connectivity (JDBC) and the Structured Query Language (SQL) to access relational databases.
java.io	Provides classes required to perform system input and output through data streams, serialization, and the file system. With the classes supplied by the java.io package you can write programs that read and write information to the console, to a file, or across a network. You can even write and read entire objects. The classes contained in this package are too numerous to mention here.
java.lang	Provides classes and interfaces that are fundamental to the design of the Java programming language. You should become intimately familiar with the functionality provided by the java.lang package. Here you will find the Object class as well as all the primitive type wrapper classes like Boolean, Character, Byte, etc. The wrapper classes come in very handy, especially when you want to convert from a String representation of a number to a primitive type. Other important classes include Thread, String, StringBuffer, Math, and Class.
java.net	Provides classes required to write networking applications. The classes supplied by the java.net package make it easy to write networked applications. Important classes contained in this package include URL, URLConnection, Socket, and ServerSocket to name just a few.
java.util	Provides a wide array of generally useful functionality like the various collection classes, date and time facilities, and utility classes like StringTokenizer, Date, Timer, Currency, and Random.
java.util.jar	Provides classes for reading and writing Java JAR files.
javax.swing	Provides a set of lightweight, all-Java components that are used to write graphical user interfaces. The objective of the javax.swing package is to provide a set of components that behave the same across all platforms. We will make extensive use of Swing components in this book

Table 5-1: Java Platform Packages Used In This Book

SUMMARY

The power of Java lies in the pre-defined functionality provided by the Java platform application programming interface (API). The Java platform API consists of a collection of packages that each contains a set of classes and interfaces ready for use in your programs.

Detailed Java platform API documentation can be found on the Java website: [java.sun.com]

The key to discovering the functionality provided by a certain class or component is to trace its inheritance hierarchy. The Java platform API documentation gives an inheritance hierarchy diagram on each class detail page. Scroll down the detail page to see what methods a class inherits from its chain of base classes.

This book will focus on and utilize a small subset of the total functionality provided by the Java platform API. However, armed with the ability to navigate an inheritance hierarchy, you can easily discover and include in your programs additional Java platform functionality not formally discussed in the text.

Skill-Building Exercises

1. **Research The Java Platform API:** Visit Sun's Java website [java.sun.com] and explore all the packages of the Java 2 Platform API.
2. **Practice Navigating An Inheritance Hierarchy:** Visit Sun's Java website and research the following classes:

java.lang.System
javax.swing.JFrame
javax.swing.JDialog
javax.swing.JApplet
java.io.InputStream
java.io.OutputStream
java.util.Date
java.util.Vector
java.net.Socket

For each class in the above list trace the complete inheritance hierarchy and draw the chain of base classes as a UML class diagram. Use figures 5-5 and 5-7 as a guide. Note the interfaces implemented by each class and conduct research on those as well. Note any deprecated methods you encounter in your research.

3. **Research Wrapper Classes:** Explore the java.lang package and research each of the primitive type wrapper classes to include: Character, Byte, Boolean, Integer, Float, and Double. Note the functionality each wrapper class provides.
4. **Research Collection Classes:** Explore the java.util package and research each of the collection classes to include: ArrayList, Vector, Stack, TreeSet, TreeMap, and HashTable.

SUGGESTED PROJECTS

1. **Procure A Java Platform API Quick Reference Guide:** Go to your favorite bookstore or library and procure one copy each of the *Java In A Nutshell* and *Java Foundation Classes In A Nutshell* books published by O'Reilly and Associates. (See the references section for more information about each of these outstanding reference books.)
2. **Study A Piece Of the Java Platform API Each Day:** This is a helpful project that will pay big dividends in the future. Concentrate each day on one or two packages and explore the classes provided therein. Study each class's constructors and methods. Trace the inheritance hierarchies and make a mental note of what functionality each package provides. You may not immediately use each and every class you encounter but when the time comes you will have a good idea of where to look for the pre-defined functionality your programs require.

SELF-TEST QUESTIONS

1. What is the Java platform?
2. What's the difference between the software development kit (SDK) and the Java runtime environment (JRE)?
3. Where's a good place to look for detailed Java platform API information?
4. In your own words explain what is meant by an inheritance hierarchy.
5. What vital information can be learned from tracing a class's inheritance hierarchy?
6. What is meant by the term deprecated?
7. List and describe the functionality provided by at least seven Java platform API packages.
8. From what common class do all Java classes (except one) inherit functionality?
9. Why do you think it's a good idea to trace a class's inheritance hierarchy?
10. What section of a class's detailed documentation page makes it easy to trace inherited functionality?

REFERENCES

Sun Inc.'s Java 2 Platform API Version 1.4.2 Website: [<http://java.sun.com/j2se/1.4.2/docs/api/index.html>]

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*, 4th Edition. O'Reilly and Associates, Inc., Sebastopol, CA. ISBN: 0-596-00283-1

David Flanagan. *Java Foundation Classes In A Nutshell: A Desktop Quick Reference*, O'Reilly and Associates, Inc., Sebastopol, CA. ISBN: 1-56592-488-6

NOTES

CHAPTER 6



STREET SCENE JAMAICA

Simple JAVA PROGRAMS: USING PRIMITIVE AND REFERENCE DATA TYPES

LEARNING OBJECTIVES

- *CREATE AND EXECUTE simple JAVA programs*
- *LIST AND DESCRIBE THE THREE CATEGORIES OF JAVA TYPES: PRIMITIVE, REFERENCE, AND ARRAY*
- *USE primitive data types in simple JAVA programs to include BOOLEAN, CHAR, BYTE, SHORT, INT, FLOAT AND double*
- *USE REFERENCE data types in simple JAVA programs*
- *USE THE NEW OPERATOR TO CREATE REFERENCE-TYPE objects*
- *DESCRIBE THE DIFFERENCES BETWEEN PRIMITIVE AND REFERENCE data types*
- *STATE THE VALUE RANGES EACH PRIMITIVE data type CAN CONTAIN*
- *LIST AND DESCRIBE THE FUNCTIONALITY PROVIDED BY THE PRIMITIVE TYPE WRAPPER CLASSES*
- *FORMULATE CORRECT AND SELF-COMMENTING identifiers*
- *LIST THE JAVA RESERVED keywords*
- *STATE THE PURPOSE OF EXPRESSIONS, STATEMENTS, AND OPERATORS*
- *LIST AND DESCRIBE THE JAVA OPERATORS*
- *EXPLAIN WHAT IT MEANS TO CALL A METHOD ON AN object*

INTRODUCTION

This chapter presents the basic concepts of Java programming. Here you will learn how to write simple Java programs using primitive and reference data types. The objective of writing simple programs is to gain a firm understanding of the structure of Java applications while at the same time building skill and confidence in your ability to create, edit, and compile Java source files and then run the resulting class files. A thorough discussion of the three different Java type categories is presented along with numerous examples of the use of primitive and reference types.

In order to master the basics of Java programming you will have to understand the concepts of statements, expressions, and operators. These will be discussed and illustrated in detail.

I will then show you how to use reference data types in your programs by introducing you to the primitive type wrapper classes. A knowledge of how these classes operate will prove handy as you progress through the text. In the process of learning about reference data types I will show you how to use the *new* operator and discuss the role of the Java virtual machine garbage collector.

The concepts and skills learned in this chapter create a strong foundation upon which you will build a greater understanding of the Java platform and the Java programming language.

Also, throughout this chapter I focus the discussion of basic language features to those found in Java version 1.4.2. The Java 5 feature of autoboxing of primitive data types is postponed until later in the book.

TERMINOLOGY

Table 6-1 defines and clarifies a few of the terms I will use in the next several sections.

Term	Definition
class	<p>The term class has two meanings: 1) a class is an object-oriented term used to denote groups of related entities (<i>things or concepts</i>) that share common traits, and 2) in Java, a class is a language construct that you use to create user-defined data types.</p> <p>A class is declared and defined within a Java source file. Java source files have a .java extension. A source file can contain many class declarations and definitions but only one of the classes can be declared to be public. The name of the source file must be the exact same name as this public class with the addition of the .java extension. Example: The class named TestClass might be defined to be:</p> <pre>public class TestClass{ }</pre> <p>The name of the source file would be TestClass.java</p>
class file	<p>When a source file is compiled with the javac compiler tool it results in one or more files that contain executable byte code. These byte code files have the extension .class. Example: When the source file TestClass.java is compiled with the javac compiler it will result in a file named TestClass.class.</p>
object	<p>The term object has several meanings as well: 1) an object is an object-oriented term used to denote entities within a problem domain, 2) in Java, an object is an instance of a class type, and 3) the word object can also mean the region of memory occupied by the data associated with an instantiated data type.</p>

Table 6-1: Terms And Definitions To Get You Started

DEFINITION OF A JAVA PROGRAM

What is a Java program? A Java program is an intentional interaction between a set of objects. One of the objects in a Java program must be capable of starting the interactions between the other objects. Three special Java object types can kick-start object interactions: 1) application objects (*applications*), 2) applet objects (*applets*), and 3) servlet objects (*servlets*). (*Servlets are part of the Java 2 Enterprise Edition (J2EE) and are therefore beyond the scope of this book, however, servlet concepts are easy to master when you understand how to write applications and applets.*)

Application Objects

A Java application object is like an ordinary Java object with one important additional feature — it contains a special method named `main()`. The `main()` method is the starting point for program execution. When you attempt to execute an application with the `java` command-line tool it expects to find a `main()` method in the class file you are trying to execute. If you attempt to execute a class file that omits the `main()` method you will receive the following error:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

The loading and execution of an application class file may result in the loading of additional class files and the creation of additional objects within the virtual machine environment. Thus, a Java program is an interaction between a set of objects.

Talking About Applications

When you talk about Java applications in your programming class or with your programmer colleagues, you will use the term “application” to refer to two things: 1) to the Java class you define in a source file that contains the `main()` method, and 2) to the collection of source files and class definitions that implement the solution to the problem you are trying to solve.

Applet Objects

Another type of Java object that can start the object interaction process is an applet object. Java applet objects are created by subclassing the `Applet` class or the `JApplet` class. Applet objects are structured differently from application objects. Applets are formally discussed in chapter 21.

CREATING SIMPLE JAVA PROGRAMS (APPLICATIONS)

To create a Java program you must define a class that contains a `main()` method, compile the source file with the `javac` command-line tool, and then run the resulting `.class` file using the `java` command-line tool. Doing so requires you to have an understanding of several skills, namely, you must:

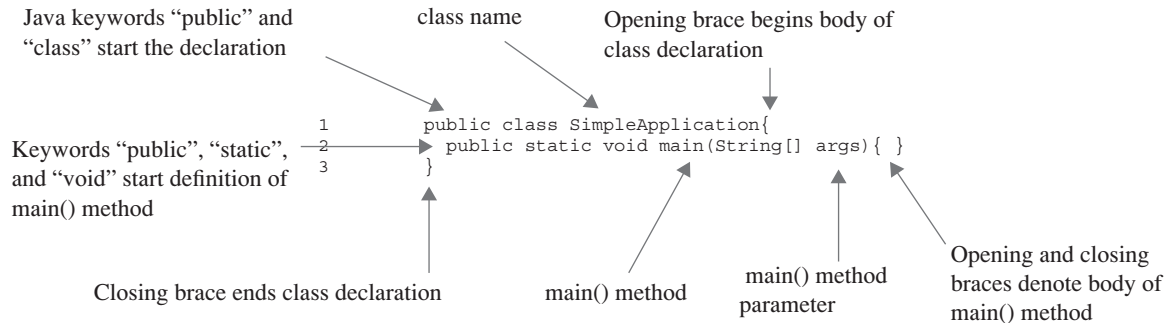
- understand how to create a source file using a text editor
- understand the basic structure of a Java class definition
- understand the structure of a Java application and the purpose of the `main()` method
- understand how to name the source file with the same name as the public class it contains
- understand how to compile the file with the `javac` compiler
- understand how to run the resulting class file with the Java virtual machine

If you encounter any difficulty along the way you will also need the qualities of tenacity and patience. You cannot be someone who gives up easily or grows impatient when things go a little wrong. The programming profession rewards those who refuse to accept defeat!

STRUCTURE OF A JAVA APPLICATION

A java application, in its simplest form, consists of a class definition that contains a main() method. Example 6.1 illustrates these points.

6.1 *SimpleApplication.java*



Although `SimpleApplication` is in fact a simple application, there’s a lot going on here that’s worth noting. First, the definition of `SimpleClass` begins on line 1 with the use of the Java keywords `public` and `class`. The meaning of each of these keywords is discussed in greater detail in the next section. For the purposes of this example these keywords are required to tell the Java compiler that we are declaring a new data type (*class*) that will be publicly accessible, meaning other classes or objects can use this class if they so require.

These keywords are followed by a string of characters that form the name of the class. The string of characters “`SimpleApplication`” is formally referred to as an identifier. Identifiers are used to form the name of classes, variables, constants, and methods. There are a few rules you must learn regarding the formation of identifier names and these rules will be covered later in this chapter.

The opening brace appearing at the end of line 1 marks the beginning of the body of the class definition. Everything appearing between the opening brace at the end of line 1 to the closing brace on line 3 belongs to the `SimpleApplication` class.

The entire `main()` method definition begins and ends on line 2. The definition of the `main()` method begins with the keywords `public`, `static`, and `void`. The `public` keyword means that `main()` can be called from other objects. The keyword `static` means the `main()` method is a class-wide method that can access static class fields. You will learn more about static and non-static methods in chapter 9. The keyword `void` indicates the `main()` method does not return a result when it completes execution.

These keywords are followed by the name of the method — `main`. The method name is followed by a set of parenthesis that can optionally contain one or more method parameters. Method parameters are used to pass data to a method when the method is called. In the case of the `main()` method it is required to take a parameter of type `String` array (`String[]`). The parameter name `args` is an ordinary identifier that could be any name. For example, you could declare your `main()` method to look like this:

```
public static void main(String[] arguments) { }
```

Notice the only thing different here is the name of the method parameter. Everything else stayed the same. For the sake of completeness, one more change could be made to the `main()` method declaration:

```
public static void main(String arguments[]) { }
```

The change here is subtle but important. Java allows you to declare arrays using C++ syntax. Notice that the brackets were moved to the right side of the parameter name. The purpose and use of the `main()` method’s `String` array is discussed in detail in chapter 8.

Returning to example 6.1, the start of the `main()` method body is indicated by the opening brace and its end is denoted by the closing brace. In this example the `main()` method body is empty. There is nothing in the body of the `main()` method and therefore when `SimpleApplication` is executed there will be no visible results.

As you can see, although `SimpleApplication` does little useful work you can learn a lot from its close examination.

Compiling and Executing SimpleApplication

Compiling and executing the SimpleApplication class is straightforward assuming you have properly set up your Java development environment. (See *chapter 2*.)

To compile the SimpleApplication source file make sure you have saved the file with the name SimpleApplication.java. Use the javac compiler tool to compile the source file as is shown here:

```
javac SimpleApplication.java
```

Notice that you must enter the complete file name, including the .java extension, when you compile the source file. This will yield a new file named SimpleApplication.class. This class file contains the byte codes that will be loaded into and executed by the Java virtual machine. To execute the SimpleApplication.class file use the java tool as is shown here:

```
java SimpleApplication
```

Note that in this case the use of the complete filename was unnecessary. The .class filename extension is assumed.

When you run SimpleApplication you will see nothing happen. That's because the main() method does nothing except start and stop. Let's expand the capability of SimpleApplication by adding a line to the main() method that will print a short text message to the console. Example 6.2 gives the source code for the modified SimpleApplication class.

6.2 SimpleApplication.java (version 2)

```
1     public class SimpleApplication{
2         public static void main(String arguments[]){
3             System.out.println("SimpleApplication lives!");
4         }
5     }
```

Figure 6-1 shows the process of compiling SimpleApplication.java and the results of executing the SimpleApplication.class file.

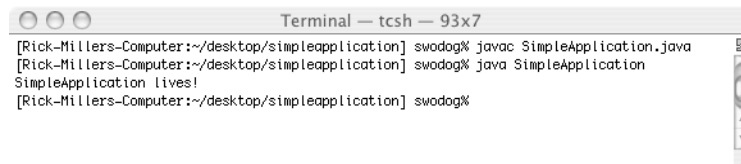


Figure 6-1: Compiling and Executing SimpleApplication

The SimpleApplication class now does some useful work. It uses the services of another Java platform API class named System. The System class provides console I/O capability ready for you to use in your programs. Here I have used the services of the System's out object and the out object's println() method to print a short text string to the console.

The important thing to take away from this example is that SimpleApplication now enlists the help of another Java class. The System class is but a tiny fraction of the pre-built functionality supplied by the Java platform classes.

Quick Review

A Java application is created by adding a main() method to an ordinary class definition. The keywords public and class are used in the class definition. The name of the class is formulated according to the Java identifier naming rules. The body of the class definition appears between the opening and closing braces. Everything in the class body belongs to the class.

The main() method appears in the class body. The name of the main() method is preceded by the keywords public, static, and void. The main() method takes a String array as its only parameter. The body of the main() method is contained between opening and closing braces.

Building Bigger Applications

Now that you know the basic structure of a Java application it's time to build upon this knowledge so that you can write applications that do more than just print simple text messages to the console. To do this you must learn more about the fundamental building blocks the Java programming language provides to help in this endeavor. The building blocks include reserved keywords, primitive, reference, and array data types, statements, expressions, operators, variables, and constants, and how to formulate variable and constant names using identifier naming rules. The rest of this chapter is devoted to covering this material. Don't feel as though you have to learn everything discussed below by heart. Some of the material is intended for future reference.

IDENTIFIERS AND IDENTIFIER NAMING RULES

Identifiers are used in Java programs to formulate the names of classes, variables, constants, and methods. There are a few rules you must be aware of regarding the correct formulation of identifier names and they are easy to remember.

- **Must Begin With Valid Java Letter** - An identifier must begin with a valid Java letter. Generally speaking, a valid Java letter is any unicode character other than the Java operators (`%`, `*`, `/`, etc.) and a few other symbols. One way to test whether or not a character you wish to use as the start of an identifier is valid is to use a handy method supplied by the `Character` class named `isJavaIdentifierStart()`. An example of its use is shown in example 6.3 below.
- **First Letter Can Be Followed By Valid Letters Or Digits** - The first letter of an identifier can be followed by any number of valid Java letters and digits. Characters to avoid are, again, the Java operators. You can use the underscore `'_'` character if you desire. If in doubt you can check the validity of the follow-on characters with another method supplied by the `Character` class named `isJavaIdentifierPart()`.
- **Identifier Cannot Spell The Name Of A Reserved Java Keyword** - The reserved Java keywords hold special meaning in the language and cannot be reused for your special purposes. The Java keywords are covered in detail in the next section.
- **Identifier Cannot Spell true, false, or null** - Although `true`, `false`, and `null` are not reserved keywords, they cannot be used as identifiers because they represent the boolean and null literals.

Example 6.3 gives the code for a short program that uses a Unicode escape sequence (`\uxxxx`) to start an identifier.

6.3 *IdentifierTest.java*

```

1  public class IdentifierTest {
2      public static void main(String[] args){
3          String \u03B63_3 = "Hello - \u03B63_3 is a valid identifier!";
4          System.out.println(\u03B63_3);
5          System.out.println(Character.isJavaIdentifierStart('\u03B6'));
6          System.out.println(Character.isJavaIdentifierStart('ô'));
7          System.out.println(Character.isJavaIdentifierStart('ç'));
8          System.out.println(Character.isJavaIdentifierStart('$'));
9          System.out.println(Character.isJavaIdentifierStart('a'));
10         System.out.println(Character.isJavaIdentifierStart('Å'));
11         System.out.println(Character.isJavaIdentifierStart('í'));
12     }
13 }
```

```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/identifier_test] swodog% java IdentifierTest
Hello - ?3_3 is a valid identifier!
true
true
true
true
true
true
true
true
```

Figure 6-2: Results of Running IdentifierTest Program

The identifier `\u03B63_3` used on line 3 actually contains two valid Java letters and two valid Java digits. The identifier begins with the Unicode escape sequence `‘\u03B6’`. This represents one Java letter. It is then followed by the character sequence `3_3` which is the Java digit `‘3’`, followed by the underscore `‘_’` character, which is then followed by the digit `‘3’` again.

On line 3 of example 6.3 the identifier `\u03B63_3` is declared to be a String reference. It is used again on line 4 as an argument to the `System.out.println()` method. This results in its String value being printed to the console.

Lines 5 through 11 of example 6.3 test several characters for their validity as identifier starting letters. The `isJavaIdentifierStart()` method returns the boolean value `true` if a letter being tested can serve as an identifier starting letter or `false` otherwise.

Figure 6-2 shows the results of running the `IdentifierTest` program. Notice how the Unicode character represented by the escape sequence `\u03B6` is represented by a `?` in the console output. That’s happening because my terminal emulator is using an encoding scheme that contains just a subset of the Unicode characters, namely, ASCII. In fact, the Unicode escape sequence is a mechanism that lets you use ASCII characters to represent Unicode characters in your programs using a standard text editor. Since most text editors only operate on the ASCII character set, the Unicode escape sequence is the only way to include Unicode characters in your Java programs.

WELL-NAMED IDENTIFIERS Will Simplify Your Life

OK, now that you know what an identifier is and the valid forms an identifier can take you will be relieved to learn that identifier naming is easy to learn. Simply avoid naming your identifiers after the reserved keywords and you’ve just about got it licked.

The most difficult part of identifier naming involves deciding on what names to use given the context of your program. The best help you can give yourself in your early programming career is to put some thought into your identifier names. Well-named identifiers will make your source code easy to read. Easy-to-read code is easier to understand. Code that’s easy to understand is easy to fix should it contain an error or two. Problems can be quickly spotted when your program reads like a story. (See *chapter 1* for a detailed discussion on an approach to identifier naming.)

JAVA RESERVED KEYWORDS

The Java programming language reserves certain identifier names for use as keywords. Any attempt by you to use a reserved keyword as an identifier will result in a compiler error. The reserved keywords, along with brief descriptions, are presented in table 6-2. Do not try to learn them by heart just yet. You will recognize several from their use in the source code examples presented in the text thus far and you will gradually be exposed to the others as you progress through the book.

Reserved Keyword	Description
abstract	Used to declare a class or method that is not completely defined.
assert	Allows you to test assumptions about how your code should behave.
boolean	Primitive Java data type representing true and false values.
break	Used to exit a containing code block. Can be used with named blocks.
case	Tests a value to select a possible execution path in a switch statement.
catch	Used in conjunction with the try keyword to trap exceptions thrown from a try/catch block.
char	Primitive Java data type representing 16-bit characters.
class	Used to declare a new class type.

Table 6-2: Reserved Java Keywords

Reserved Keyword	Description
<i>const</i>	Reserved, but not currently used in the language.
continue	Used inside iteration statements <i>for</i> , <i>while</i> , and <i>do</i> to quit the current iteration and start a new one.
default	Denotes the default case in a switch statement.
do	Iteration statement that executes a code block at least once.
double	Primitive Java data type denoting 8-byte floating point numeric values.
else	Used in conjunction with the if statement to specify an alternative execution path.
enum	Used to declare enumerated types in Java 5
extends	Used in a class declaration to specify a base class from which to inherit functionality or behavior.
final	Used to declare Java variables that cannot be changed (constants) and in method and class declarations to prevent overriding and subclassing.
finally	Used in conjunction with try/catch blocks to specify a block of code that must always be executed whether or not an exception is thrown.
float	Primitive Java data type denoting 4-byte floating point numeric values.
for	Iteration statement used to execute a block of code a specified or indefinite number of times.
<i>goto</i>	Reserved, but not currently used in the language.
if	Allows conditional execution of a block of code based on the evaluation of an expression. Can contain an else clause that provides an alternative execution path.
implements	Used in class declarations to specify what interfaces the class adopts.
import	Used to provide abbreviated name access to classes contained in packages.
instanceof	A Java operator used to test the types of objects.
int	Primitive Java data type denoting 32-bit integer values.
interface	Used to declare an interface type.
long	Primitive Java data type denoting 64-bit integer values.
native	Used to declare methods that will have an implementation in a language other than Java.
new	Used to allocate memory for objects during program execution.
package	Used to specify what package name a class belongs too.
private	Visibility modifier. Used to specify class, member, or attribute visibility.
protected	Visibility modifier. Used to specify class, member, or attribute visibility.
public	Visibility modifier. Used to specify class, member, or attribute visibility.
return	Used to return a value from a method.
short	Primitive Java data type denoting 16-bit integer values.
static	Used to declare class-wide methods or attributes.

Table 6-2: Reserved Java Keywords

Reserved Keyword	Description
strictfp	Rarely used modifier. Used to specify that all floating point computation performed by a method must conform to the IEEE 754 standard.
super	Used to gain access to a class's immediate base class. Used in three ways: 1) to call base class constructors from derived class constructors, 2) to call a base class method vs. its overridden method, and 3) to access a base class attribute that's hidden by a subclass attribute of the same name.
switch	A flow control statement used to provide several alternative execution paths. Used with the case and default keywords. Used in place of multiple nested if/else statements where appropriate.
synchronized	Used as a statement and modifier. Specifies that code belonging to an object, array, class, or method must be treated as a critical section and not accessed simultaneously by different threads.
this	Used to explicitly indicate same-class field and method access.
throw	Used to explicitly raise an exception condition.
throws	Used in a method definition to specify what, if any, exceptions may result from its execution.
transient	A field modifier that specifies a field is not part of the persistent state of an object and therefore should not be serialized.
try	Introduces a block of code that, when executed, may result in a checked-exception being thrown. Used in conjunction with the catch and finally keywords.
void	Used to specify that a method does not return a value.
volatile	Field modifier specifying that the field may be accessed by unsynchronized threads.
while	Iteration statement that will evaluate an expression prior to executing a code block. The expression may evaluate to false which would result in the code block not being executed.
true, false, null	These are not reserved keywords but can be thought of as such.

Table 6-2: Reserved Java Keywords

JAVA TYPE CATEGORIES

The Java language supports two distinct type categories: primitive and reference. (*Arrays are a special case of reference data type and have unique features that makes it easy to think of them as being their own distinct type category.*) Any variables or constants you use in your programs will belong to one of these categories. It's important to know how the types in each category behave. (*The concepts of variables and constants are discussed in detail later in this chapter.*)

PRIMITIVE DATA TYPES

Primitive type names begin with lowercase letters and DO NOT require memory to be dynamically allocated with the new operator. Table 6-3 lists the primitive data types along with their value ranges.

Primitive Type	Size in Bytes	Default Value	Value Range
boolean	1 Bit	false	true or false
char	2	\u0000	\u0000 to \uFFFF

Table 6-3: Java Primitive Data Types

Primitive Type	Size in Bytes	Default Value	Value Range
byte	1	0	-128 to 127
short	2	0	-32768 to 32767
int	4	0	-2147483648 to 2147483647
long	8	0	-9223372036854775808 to 9223372036854775807
float	4	0.0	$\pm 1.4e^{-48}$ to $\pm 3.4028235e^{+38}$
double	8	0.0	$\pm 4.9e^{-324}$ to $\pm 1.7976931348623157e^{+308}$

Table 6-3: Java Primitive Data Types

REFERENCE DATA TYPES

Anything not a primitive data type is a reference data type. Reference data types include all the classes of the Java platform and any classes you create as part of your programs. Generally speaking, reference data types begin with capital letters, but you could define classes that begin with lowercase letters if you set your mind to it.

Before using a reference-type object you must first allocate memory space for it using the *new* operator. The exception to this rule is the String class which holds a special place among Java platform reference types because of the shortcut method you can use to assign values to String objects.

ARRAY TYPES

An array is a data structure that's used to store a collection of homogeneous primitive data type values or references to objects in a contiguous memory allocation. (*In other words, an array holds like-sized items arranged one after the other!*)

Arrays are reference types in the Java language. This gives them special powers when compared to arrays in programming languages like C or C++. Because arrays are treated like first-class objects they provide additional functionality you will find extremely helpful. Arrays are covered in detail in chapter 8.

WORKING WITH PRIMITIVE TYPES

This section provides a detailed discussion of how to use Java primitive data types in your programs. I will begin by showing you how to declare and use primitive data type variables and constants in the main() method. I will then show you how to declare and use class and instance variables. It's important to understand how the main() method and objects declared within its body relate to the enclosing class structure. Learning this stuff now will avoid a lot of confusion later.

During this discussion I will use several Java language operators to perform value assignments and simple arithmetic on the constants and variables. These operators include '=' for assignment, and '+' for addition. Other Java operators are discussed in detail later in the chapter.

APPLICATION CLASS STRUCTURE

Example 6.4 presents the source code for the application class named TestClassOne that I will use to present the concepts of primitive-type variables and constants.

```

1     public class TestClassOne {
2
3         public static void main(String[] args){
4
5

```

6.4 TestClassOne.java

```

6     }
7     }

```

Before getting started I'd like to present you with a good working definition of the term variable.

DEFINITION OF THE TERM “VARIABLE”

A variable is a named location in memory whose value can be changed during the course of program execution.

DECLARING AND USING VARIABLES IN THE MAIN() METHOD

Before you can use a variable in a Java program you must declare the variable by specifying its data type and its name. Example 6.5 shows an integer (int) variable being declared and used in the TestClassOne main() method.

6.5 TestClassOne.java (mod 1)

```

1     public class TestClassOne {
2
3         public static void main(String[] args){
4             int int_variable = 0;
5             System.out.println(int_variable);
6         }
7     }

```

The name of the variable is int_variable and the value being assigned to it is zero. The variable is then being used as an argument to the System.out.println() method which prints its value to the console as shown in figure 6-3.

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/testclass] swodog$ java TestClassOne
0
[Rick-Millers-Computer:~/desktop/testclass] swodog$

```

Figure 6-3: TestClassOne Mod 1 Output

Let's add another variable declaration to the main() method as shown in example 6.6.

6.6 TestClassOne.java (mod 2)

```

1     public class TestClassOne {
2
3         public static void main(String[] args){
4             int int_variable = 0;
5             int int_variable_2 = int_variable;
6
7             System.out.println(int_variable);
8             System.out.println(int_variable_2);
9         }
10    }

```

In this example another integer variable named int_variable_2 is declared and its value is initialized using the value of int_variable. Both their values are then printed to the console by the System.out.println() method. Notice how primitive type values are assigned directly to the variable name using the assignment operator =.

Example 6.7 adds a few more variables to the main() method and uses them to perform several arithmetic operations.

6.7 TestClassOne.java (mod 3)

```

1     public class TestClassOne {
2
3         public static void main(String[] args){
4             int int_variable = 0;
5             int int_variable_2 = int_variable;
6             int total = 0;
7
8             int_variable = 2;
9             int_variable_2 = 3;
10            total = int_variable + int_variable_2;
11
12
13            System.out.println(int_variable);
14            System.out.println(int_variable_2);

```

```

15         System.out.println(total);
16         System.out.println(int_variable + int_variable_2);
17     }
18 }

```

Figure 6-4 shows the output that results from running this program.

```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/testclass] swodog% java TestClassOne
2
3
5
5
[Rick-Millers-Computer:~/desktop/testclass] swodog%

```

Figure 6-4: TestClassOne Mod 3 Output

Referring to example 6.7 above, notice how the variable declarations are grouped toward the top of the main() method. The variable named int_variable is declared first and is then used on the next line to initialize int_variable_2. This illustrates an important point — you must declare and initialize a variable before you can use it in your program. Once declared, it is available for use on the next line.

On lines 8 and 9 the values of int_variable and int_variable_2 are changed to 2 and 3 respectively. On line 10 the new values of int_variable and int_variable_2 are used to initialize the variable named total. The values of all three variables are then printed to the console as shown in figure 6-4 above.

Notice on line 16 that int_variable and int_variable_2 are added together within the parentheses of the System.out.println() method. This is acceptable because the result of the addition yields an integer value which is recognized by the println() method.

DEFINITION OF THE TERM “CONSTANT”

A constant is a named location in memory whose value, once set, cannot be changed during the course of program execution. A constant, in the Java programming language, is a variable that’s declared to be final. (*i.e.*, a *final variable*)

DECLARING AND USING CONSTANTS IN THE MAIN() METHOD

Constant declarations begin with the final keyword. Their value must be initialized at the point of declaration and once set, cannot be changed during program execution. Example 6.8 shows a constant being declared and used in the TestClassOne main() method.

```

1     public class TestClassOne {
2
3         public static void main(String[] args){
4             final int CONST_VALUE = 25;
5             System.out.println(CONST_VALUE);
6         }
7     }

```

6.8 TestClassOne.java (mod 4)

Figure 6-5 shows the results of running this program.

```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/testclass] swodog% java TestClassOne
25
[Rick-Millers-Computer:~/desktop/testclass] swodog%

```

Figure 6-5: TestClassOne Mod 4 Output

You can use a variable or another constant to initialize a constant’s value as is shown by example 6.9.

```

1     public class TestClassOne {
2         public static void main(String[] args){
3             final int CONST_VALUE = 25;
4             final int CONST_VALUE_2 = CONST_VALUE;

```

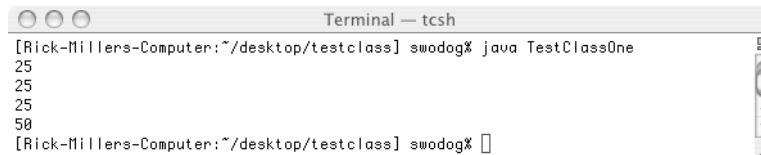
6.9 TestClassOne.java (mod 5)

```

5         int int_variable = CONST_VALUE_2;
6         final int CONST_VALUE_3 = int_variable + CONST_VALUE_2;
7         System.out.println(CONST_VALUE);
8         System.out.println(CONST_VALUE_2);
9         System.out.println(int_variable);
10        System.out.println(CONST_VALUE_3);
11    }
12 }

```

Figure 6-6 shows the results of running this program.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/testclass] swodog% java TestClassOne
25
25
25
50
[Rick-Millers-Computer:~/desktop/testclass] swodog%

```

Figure 6-6: TestClassOne Mod 5 Output

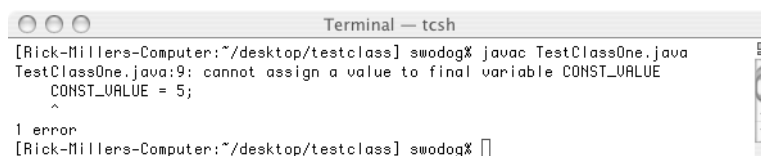
Referring to example 6.9 above, notice how the constants are declared using all uppercase letters. Doing so sets them apart from variables and makes them easy to spot in the code. Constants, once declared and initialized, can be used like variables in your source code as is illustrated on line 6. The only difference is that any attempt to change a constant's value after it has been declared and initialized will result in a compiler error. This is illustrated by the code shown in example 6.10 and the compiler error message shown in figure 6-7.

6.10 TestClassOne.java (mod 6)

```

1     public class TestClassOne {
2         public static void main(String[] args){
3             final int CONST_VALUE = 25;
4             final int CONST_VALUE_2 = CONST_VALUE;
5                 int int_variable = CONST_VALUE_2;
6             final int CONST_VALUE_3 = int_variable + CONST_VALUE_2;
7
8             CONST_VALUE = 5; // This will result in a compiler error
9
10            System.out.println(CONST_VALUE);
11            System.out.println(CONST_VALUE_2);
12            System.out.println(int_variable);
13            System.out.println(CONST_VALUE_3);
14        }
15    }

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/testclass] swodog% javac TestClassOne.java
TestClassOne.java:9: cannot assign a value to final variable CONST_VALUE
    CONST_VALUE = 5;
    ^
1 error
[Rick-Millers-Computer:~/desktop/testclass] swodog%

```

Figure 6-7: Compiler Error Message Resulting From Attempt To Change A Constant's Value

GETTING SIMPLE INPUT INTO YOUR PROGRAM

Up to now I've shown you how to set the value of variables and constants in short programs and use those variables and constants to perform simple arithmetic operations. However, it would be nice to input values into the program when it starts running. Unfortunately, although Java makes it easy to print output to the console, it's not so easy to accept input from the console without first digging deeper into the Java platform classes for help.

To avoid this I will take a different approach. I will show you how to use the main() method's String array to get input from the console when the program first starts. This will tide you over until you formally learn how to create reference type objects in the next section.

Using The main() Method's String Array To Accept Input At Program Startup

When a Java application is started from the console a series of arguments can be input into the program by way of the main() method's String array. The name of the array parameter is usually named args which is short for argu-

ments. Example 6.11 gives the source code for `TestClassOne.java` showing how the `String` array is used to accept three `String` values from the command line at program startup.

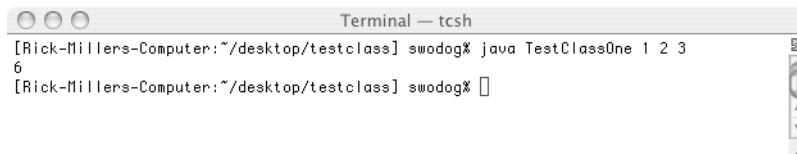
6.11 `TestClassOne.java` (mod 7)

```

1      public class TestClassOne {
2
3          public static void main(String[] args){
4              int int_variable_1 = Integer.parseInt(args[0]);
5              int int_variable_2 = Integer.parseInt(args[1]);
6              int int_variable_3 = Integer.parseInt(args[2]);
7
8              int total = int_variable_1 + int_variable_2 + int_variable_3;
9
10             System.out.println(total);
11
12         }
13     }

```

This example also employs the services of a primitive type wrapper class named `Integer`. Let's take a look at the program in action and then discuss what's happening in the code. Figure 6-8 shows the results of running this program.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/testclass] swodog% java TestClassOne 1 2 3
6
[Rick-Millers-Computer:~/desktop/testclass] swodog%

```

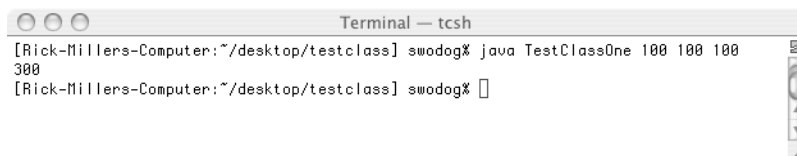
Figure 6-8: Results of Running `TestClassOne` with the Input 1 2 3

As figure 6-8 illustrates, the program is run from the command line with the `java` tool as usual, however, three digits follow the name of the class. These three digits are read into the `main()` method `String` array as string arguments and are converted into integer values by the `Integer.parseInt()` method.

Each position of the `args` array is accessed with the subscript operator `[]` and an integer value. The `String` array element `args[0]` represents the first element of the array and contains the value "1". This is used on line 4 as an argument to the `Integer.parseInt()` method. The second element of the `args` array, `args[1]`, contains the value "2", and the third element, `args[2]`, contains the value "3". These are converted into `int` values with the `Integer.parseInt()` method on lines 5 and 6.

Once the `Strings` have all been converted to integer values the variables are added and the result is assigned to the `total` variable. The value of the `total` variable is then printed to the console as is shown by figure 6-8.

This program is a slight improvement upon previous programs but is still rather clunky because you must always enter three numbers to add — no more — no less. It will also cause a runtime error if one of the values supplied to the program on the command line does not translate into an integer. Figure 6-9 shows the results of running the program again with three different values.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/testclass] swodog% java TestClassOne 100 100 100
300
[Rick-Millers-Computer:~/desktop/testclass] swodog%

```

Figure 6-9: Running `TestClassOne` Again with Different Input Values

In order to write more powerful programs you will have to master the use of reference data types.

WORKING WITH REFERENCE TYPES

Reference data types behave differently from primitive data types. As you saw in the previous section, to use a primitive data type variable in a program you simply declare the variable and assign it a value directly. You can't do this with reference data types. (*You can as of Java version 1.5. This is referred to as `autoboxing`. But I will delay coverage of `autoboxing` until later in the book.*) There's another step involved that requires the use of the `new` operator to

allocate memory, create an object, and assign the value of the object's memory location to the reference identifier. Only after the object is created in memory can you then use the reference to access the object's functionality. The following sections discuss this process in detail.

DEFINITION OF A REFERENCE VARIABLE

A reference variable is an identifier that can be assigned the value of an object's memory location during program runtime. Once the value of an object's memory location has been assigned to the reference it can be used to access the object's functionality. Once you know how to use reference variables the whole Java platform API is at your service.

DEFINITION OF AN OBJECT

An object is a region of memory that contains the data associated with a particular instance of a class. The object, or instance, is created when the *new* operator is used to dynamically allocate system memory for the object's use. The act of creating an object is referred to as *instantiation*.

CREATING OBJECTS WITH THE NEW OPERATOR

Example 6.12 gives the source code for a class named `TestClassTwo` showing how to use the `new` operator to instantiate an object and assign its memory location value to a reference variable.

6.12 *TestClassTwo.java*

```

1     public class TestClassTwo {
2         public static void main(String[] args){
3             Object object_reference = new Object();
4             System.out.println(object_reference.toString());
5         }
6     }

```

In this example I am creating an object of type `java.lang.Object`. A reference variable named `object_reference` is declared on line 3. On the same line the `new` operator is used to create the object in memory and assign its location value to `object_reference`. Notice that the `new` operator is followed by the class name of the object you want to create. In this case the name of the class is `Object`. Appended to the `Object` class name is a set of parentheses. This is referred to as a *constructor call*. A constructor is a special method whose purpose is to properly set up or construct an instance of an object in memory. You'll learn more about constructors in chapter 9.

After the object is created in memory its functionality can be accessed via the reference variable as is shown on line 4. In this case, the `toString()` method is being called via the `object_reference` variable. The `toString()` method returns a `String` representation of the object which includes its name followed by a hexadecimal value that represents the object's *hashcode*. (*An object's hashcode is an integer value that is unique to that particular object.*) This string is then used as an argument to the `System.out.println()` method. Figure 6-10 shows the results of running this program.

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/testclasstwo] swodog% java TestClassTwo
java.lang.Object@8813f2
[Rick-Millers-Computer:~/desktop/testclasstwo] swodog%

```

Figure 6-10: Results of Running Example 6.12

Figure 6-11 shows graphically what happens when an object is instantiated with the `new` operator. Memory space for dynamically created objects is reserved in an area of memory referred to as the *heap*. Once space has been reserved on the heap an integer value representing the object's memory location is returned to the requesting program and assigned to the `object_reference` variable. (*An object's memory address may be used to compute its hashcode value but this is not strictly required.*) In the case of example 6.12 above, when that program was run on my computer the newly created object had a hashcode value of 8813f2 as is shown in figure 6-10 above.

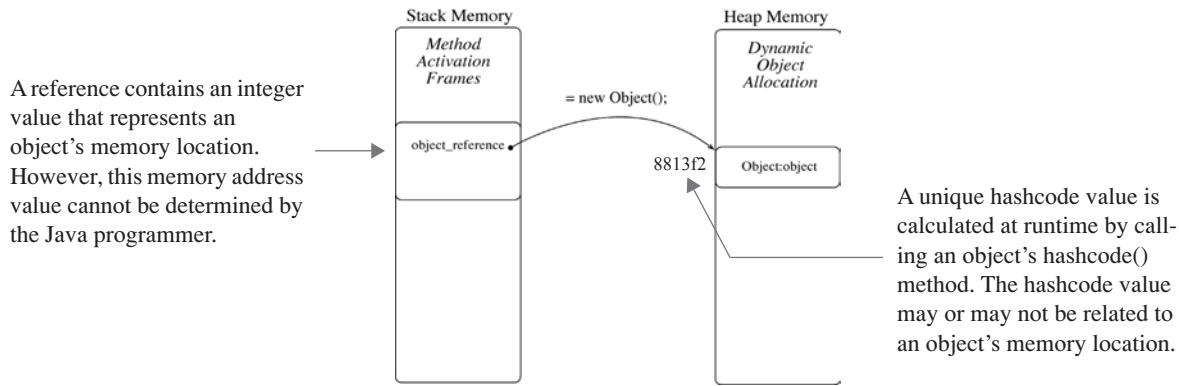


Figure 6-11: Creating an Object with the *new* Operator

GARBAGE COLLECTION

When you want to create a new object and assign its memory location to an existing reference variable just go ahead and use the new operator again. The Java virtual machine will manage the memory issues for you. Using what is referred to as a *garbage collector*, the Java virtual machine will periodically check the objects in the heap to ensure they have valid references pointing to them. If not, they are tagged for collection and the memory space they occupy is freed up for use by other objects. Example 6.13 shows the object_reference variable being reused to hold the location of a new object.

6.13 TestClassTwo.java (mod 2)

```

1      public class TestClassTwo {
2          public static void main(String[] args){
3              Object object_reference = new Object();
4              System.out.println(object_reference.toString());
5              object_reference = new Object();
6              System.out.println(object_reference.toString());
7          }
8      }
    
```

Notice that on line 5 the new operator is being used to create another Object class object on the heap. The new object's memory location value will overwrite the old value stored in the object_reference variable and it will now point to, or reference, the newly created object. Figure 6-12 shows the results of running this program.

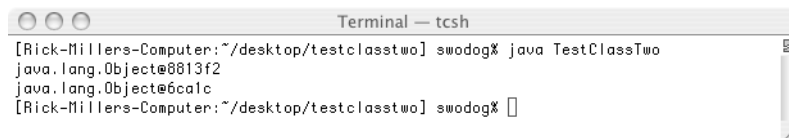


Figure 6-12: Creating Another Object with the new Operator

Notice now that the second object has a completely new memory location as is shown in the program output. Figure 6-13 graphically illustrates what's happening here.

Now that the object_reference variable points to a new object, the old object it used to point to no longer has a reference pointing to it. It can now be garbage collected and its memory reclaimed for use by new objects as required.

ACCESSING CLASS MEMBERS FROM THE MAIN() METHOD

Although you will not be formally introduced to classes until chapter 9, it's a good idea to show you the relationship between a class's variables and constants and the main() method. A clear understanding of this relationship early in your Java programming career will save you a lot of headaches.

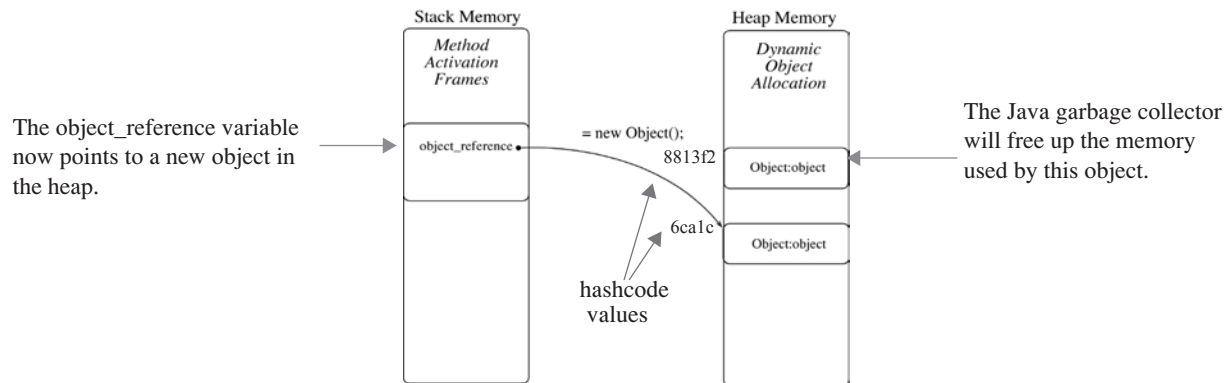


Figure 6-13: Reusing the object_reference Variable

THE MAIN() METHOD IS A STATIC METHOD

The main() method is a static method. Say this to yourself about a hundred times! “The main() method is a static method.” Because it is a static method it can only directly access class variables or constants that are declared to be static.

DEFINITION OF THE TERM CLASS VARIABLE/CONSTANT

A variable or constant declaration, appearing in the body of a class definition outside of any method, declared to be static, is referred to as a class variable or constant, meaning the values they contain are available to all dynamically created objects of that class type.

DEFINITION OF THE TERM INSTANCE VARIABLE/CONSTANT

A variable or constant declaration, appearing in the body of a class definition outside of any method, declared to be non-static, is referred to as an instance variable or constant, meaning there is a copy of each of those variables or constants for each dynamically created object. To access an instance variable or constant from a static method requires a reference to an object.

AN EXAMPLE

Example 6.14 gives the code for a class named TestClassThree. TestClassThree contains two static and two instance variable declarations.

6.14 TestClassThree.java

```

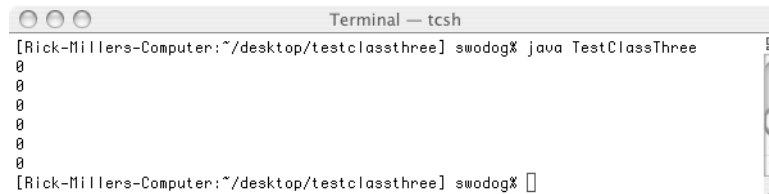
1      public class TestClassThree {
2          public static int static_int_variable;           // main() can access this variable
3          private static int static_int_variable_2;       // and this one too...
4
5          public int instance_int_variable;               // But it needs a reference to an object
6          private int instance_int_variable_2;           // to access these instance variables...
7
8          public static void main(String[] args){
9              System.out.println(static_int_variable);
10             System.out.println(static_int_variable_2);
11             System.out.println(TestClassThree.static_int_variable);
12             System.out.println(TestClassThree.static_int_variable_2);
13
14             TestClassThree tc3 = new TestClassThree();
15
16             System.out.println(tc3.instance_int_variable);
17             System.out.println(tc3.instance_int_variable_2);
18         }
19     }

```


As you can see by examining the code, the `main()` method can directly access the static variables as is shown on lines 9 and 10. Lines 11 and 12 shows an alternative, fully-class-name-qualified, way of accessing `TestClassThree`'s static variables. You will see this access method used a lot in Java programs. Notice too that because the `main()` method belongs to the `TestClassThree` class, it can directly access private class variables as well.

However, for the static `main()` method to access the instance or non-static `TestClassThree` variables it must have a reference to a `TestClassThree` object. This is provided on line 14. The name of the reference variable is `tc3` and is used to access each of the instance variables as is shown on lines 16 and 17.

Figure 6-14 shows the results of running example 6.14. Notice that all variable values are zero. This is the default value of integer type class and instance variables.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/testclassthree] swodog% java TestClassThree
0
0
0
0
0
0
[Rick-Millers-Computer:~/desktop/testclassthree] swodog% 
```

Figure 6-14: Results of Running Example 6.14

STATEMENTS AND EXPRESSIONS

Java programs contain a series of statements and expressions. For the most part these statements are terminated by a semicolon. In fact, one type of valid statement in the Java programming language is the empty statement:

```
;
```

The *empty statement* is simply a semicolon. It could appear on a line by itself but it doesn't have to. Two statements could appear on the same line like so:

```
;;
```

The following statement is known as an expression statement:

```
int variable_name = 3;
```

A series of statements can be contained within a block. A block of statements is contained within a set of curly braces. Study the code shown in example 6.15.

6.15 *StatementTest.java*

```
1     public class StatementTest {
2
3         public static void main(String[] args){
4             ;
5             ;;
6             enclosed_block_1: {
7                 int variable_name = 3;
8                 System.out.println(variable_name);
9             }
10
11            enclosed_block_2: {
12                int variable_name = 5;
13                System.out.println(variable_name);
14            }
15        } //end main() method body
16    } // end StatementTest class definition
```

The `StatementTest` class contains a `main()` method. The statements comprising the `main()` method are contained within its body which is denoted by the opening brace appearing at the end of line 3 and the closing brace appearing on line 15.

The `main()` method contains several types of statements. Line 4 has one empty statement and line 5 has two empty statements. Line 6 is interesting because it provides a labeled statement block named `enclosed_block_1`. The `enclosed_block_1` statement block body is contained between its opening brace on line 6 and its closing brace on line 9. All variables declared and used within that block of code are said to be local to that block of code.

Line 11 begins another labeled statement block named `enclosed_block_2`. Notice that another variable named `variable_name` is declared and used within that statement block. Figure 6-15 shows the results of running example 6.15.

```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/statementtest] swodog% java StatementTest
3
5
[Rick-Millers-Computer:~/desktop/statementtest] swodog%
```

Figure 6-15: Results of running Example 6.15

You will be gradually introduced to the other types of Java statements and expressions as you progress through the text. But keep example 6.15 in mind because the points it illustrates will be very helpful to you when you learn how to declare class methods that contain local variables.

OPERATORS

Java statements and expressions usually contain some form of Java operator. You've been exposed to several operators already, namely, the assignment operator, the addition operator, and the new operator (`=`, `+`, `new`). Table 6-4 lists the operators and gives a brief description of each. You will be gradually exposed to most of these operators as you progress through the text.

HOW TO READ THE OPERATOR TABLE

The operators are listed by descending precedence order. The higher the precedence number the tighter the operator binds to its operands. Some operators associate left to right while others associate right to left. The operands column specifies what type of operands the operator expects. Some operators are overloaded to perform a similar operation on different types of operands.

Operator	Precedence	Associativity	Operands	Description
<code>++</code>	15	Left to Right	variable	postfix increment operator
<code>--</code>	15	Left to Right	variable	postfix decrement operator
<code>(args)</code>	15	Left to Right	method, argument list	method invocation
<code>[]</code>	15	Left to Right	array, int	array element access
<code>.</code>	15	Left to Right	object, member	object member access
<code>!</code>	14	Right to Left	boolean	boolean NOT operator
<code>~</code>	14	Right to Left	int	bitwise complement
<code>+</code>	14	Right to Left	number	unary plus operator
<code>-</code>	14	Right to Left	number	unary minus operator
<code>++</code>	14	Right to Left	variable	prefix increment operator
<code>--</code>	14	Right to Left	variable	prefix decrement operator
<code>(type)</code>	13	Right to Left	type, any	type conversion (cast) operator
<code>new</code>	13	Right to Left	class, argument list	object creation operator

Table 6-4: Java Operators

Operator	Precedence	Associativity	Operands	Description
*	12	Left to Right	number, number	multiplication operator
/	12	Left to Right	number, number	division operator
%	12	Left to Right	int, int	integer remainder (modulus) operator
+	11	Left to Right	number, number	addition operator
-	11	Left to Right	number, number	subtraction operator
+	11	Left to Right	string, any	string concatenation operator
<<	10	Left to Right	int, int	left shift operator
>>	10	Left to Right	int, int	right shift with sign extension operator
>>>	10	Left to Right	int, int	right shift with zero extension operator
instanceof	9	Left to Right	reference, type	type comparison operator
<	9	Left to Right	number, number	less than operator
<=	9	Left to Right	number, number	less than or equal to operator
>	9	Left to Right	number, number	greater than operator
>=	9	Left to Right	number, number	greater than or equal to operator
!=	8	Left to Right	reference, reference	not equal (different objects) operator
==	8	Left to Right	reference, reference	equality (the same object) operator
!=	8	Left to Right	primitive, primitive	not equal (different values) operator
==	8	Left to Right	primitive, primitive	equality (same value) operator
&	7	Left to Right	boolean, boolean	boolean AND operator
&	7	Left to Right	int, int	bitwise AND operator
^	6	Left to Right	boolean, boolean	boolean XOR operator
^	6	Left to Right	int, int	bitwise XOR operator
	5	Left to Right	boolean, boolean	boolean OR operation
	5	Left to Right	int, int	bitwise OR operator
&&	4	Left to Right	boolean, boolean	conditional AND operator
	3	Left to Right	boolean, boolean	conditional OR operator
?:	2	Right to Left	boolean, any, any	ternary conditional operator
*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, =	1	Right to Left	variable, any	Combines assignment with operation
=	1	Right to Left	variable, any	assignment

Table 6-4: Java Operators

USE PARENTHESES TO EXPLICITLY FORCE PRECEDENCE

In some cases it will be clear how you intend an operator to be applied, associated, and evaluated in an expression. Take, for example, the following piece of code:

```
some_variable = 4 + 6;
```

In this case the expression $4 + 6$ will be evaluated first to yield a value of 10 which is then assigned to `some_variable`. However, consider for a moment the following expression:

```
some_variable = 7 * 8 / 3 + 22 - 38;
```

A strict evaluation according to the precedence of each operator will yield one value being assigned to `some_variable` but the application of parentheses will yield another:

```
some_variable = 7 * (8 / (3 + (22 - 38)));
```

Using parentheses to explicitly force operator association makes your code easier to read and reduces the possibility of making a mistake in an expression.

JAVA OPERATORS IN DETAIL

This section presents a detailed discussion of most of the Java operators. Those not covered below will be discussed in greater detail later in the text. It's important to know not only what kind of operators are available for your use but also what, if any, limitations or concerns you should be aware of when you use an operator.

ARITHMETIC OPERATORS $+$, $-$, $*$, $/$

The arithmetic operators perform the familiar operations of addition, subtraction, multiplication, and division. They take as operands all the numeric primitive data types. These include byte, short, int, long, float, and double.

You should remember at all times that you are not adding, subtracting, multiplying, or dividing actual numbers, but machine representations of numbers. Each primitive numeric data type has a range of authorized values. If you are not careful you may find that your programs behave in unexpected ways because you have unknowingly exceeded the range of a particular type.

Pay particular attention to operator precedence. The multiplication `*` and division `/` operators take precedence over addition `+` and subtraction `-`. I recommend using parentheses to explicitly force the particular order of evaluation you desire. Doing so benefits you rather than the computer as it forces you to think about the code you are writing.

Another aspect of usage to consider is when operators are applied to different data types in the same expression. For example, the byte data type holds a very small positive and negative value when compared with the int or long data type. The float and double data types store decimal values but the integer types do not. The capability of one numeric data type to store a larger value than another makes that larger data type more precise. Any attempt to assigned the value of a larger data type to a smaller, less precise, data type, will result in a compiler error warning of a possible loss of precision.

When you are performing division and expect a very small positive decimal value you should use the float or double data types. Study the code shown in example 6.16 below and compare it with its output shown in figure 6-16.

6.16 *DivisionTest.java*

```

1      public class DivisionTest {
2
3          public static void main(String[] args){
4              int int_i = 0;
5              int int_j = 1;
6              int int_k = 320;
7
8              float float_i = 0.0f;
9              float float_j = 1.0f;
10             float float_k = 320.0f;
11
12             int_i = int_j/int_k;
13             System.out.println(int_i);
14
15             float_i = int_j/int_k;
16             System.out.println(float_i);
17
18             /***** generates loss of precision error *****/
19             int_i = int_j/float_k;
```

```

20     System.out.println(int_i);
21     *****/
22
23     float_i = int_j/float_k;
24     System.out.println(float_i);
25
26     float_i = float_j/int_k;
27     System.out.println(float_i);
28
29     }
30     }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/divisiontest] swodog% java DivisionTest
0
0.0
0.003125
0.003125
[Rick-Millers-Computer:~/desktop/divisiontest] swodog%

```

Figure 6-16: Results of Running Example 6.16

Referring to example 6.16, several division operations are performed with a combination of int and float data type variables. The value 1 is divided by the value 320. On line 12 all data types involved in the expression are int. This results in the value 0 being assigned to the variable int_i. Line 15 produces the same result even though the receiving variable is a float.

Lines 23 and 26 produce the desired result. When one of the division operands is a floating point type the result of the division is automatically converted (cast) by the compiler to a floating point type. If the receiving variable is a floating point type then all will go well. If not, as is the case with line 19, then a compiler error will result warning of a possible loss of precision.

STRING CONCATENATION OPERATOR +

The plus symbol is also used in Java to perform String concatenation. The String concatenation operator is used to create a new String by joining one or more Strings or by joining a String with another data type. The other data type will be converted to a String and then combined with the existing String value to form a new String object. Example 6.17 offers a few examples of its use.

6.17 StringOperatorTest.java

```

1     public class StringOperatorTest {
2         public static void main(String[] args){
3             String salutation = "Hello ";
4             String welcome_msg = "Welcome to Java For Artists";
5             String name      = args[0];
6
7             String complete_msg = salutation + args[0] + ", " + welcome_msg + '!';
8
9             System.out.println(complete_msg);
10        }
11    }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/StringOperatorTest] swodog% java StringOperatorTest reader
Hello reader, Welcome to Java For Artists!
[Rick-Millers-Computer:~/desktop/StringOperatorTest] swodog%

```

Figure 6-17: Results of Running Example 6.17

Example 6.16 takes a command-line argument String and concatenates it with the other program String variables to form a complete welcome message that is displayed when the program is executed.

MODULUS (INTEGER REMAINDER) OPERATOR %

The modulus operator performs integer division and returns the remainder. Example 6.18 shows the modulus operator in action. Figure 6-18 shows the results of running this program.

6.18 *ModulusOperatorTest.java*

```

1      public class ModulusOperatorTest {
2          public static void main(String[] args){
3              System.out.println(100 % 1);
4              System.out.println(100 % 2);
5              System.out.println(100 % 3);
6              System.out.println(100 % 4);
7              System.out.println(100 % 6);
8              System.out.println(100 % 11);
9              System.out.println(100 % 12);
10             System.out.println(100 % 13);
11             System.out.println(100 % 23);
12             System.out.println(100 % 27);
13             System.out.println(100 % 100);
14         }
15     }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/modulusoperatortest] swodog% java ModulusOperatorTest
0
0
1
0
4
1
4
9
8
19
0
0
0
[Rick-Millers-Computer:~/desktop/modulusoperatortest] swodog%

```

Figure 6-18: Results of Running Example 6.18

GREATER-THAN/LESS-THAN OPERATORS >, >=, <, <=

The greater-than and less-than operators, including the greater-than-or-equals-to and less-than-or-equals-to operators, operate on numeric primitive data types and return a boolean value of either true or false. Example 6.19 shows these operators in action.

6.19 *GreaterLessThanTest.java*

```

1      public class GreaterLessThanTest {
2          public static void main(String[] args){
3              System.out.println(2 > 1);
4              System.out.println(2 < 1);
5              System.out.println(2 >= 1);
6              System.out.println(2 <= 1);
7              System.out.println(2 > 2);
8              System.out.println(2 < 2);
9              System.out.println(2 >= 2);
10             System.out.println(2 <= 2);
11         }
12     }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/greaterlessthanoptest] swodog% java GreaterLessThanTest
true
false
true
false
false
false
true
true
true
true
[Rick-Millers-Computer:~/desktop/greaterlessthanoptest] swodog%

```

Figure 6-19: Results of Running Example 6.19

EQUALITY AND INEQUALITY OPERATORS ==, !=

The equality and inequality operators can be applied to primitive and reference data types. When applied to primitive types the operators compare actual values. When applied to reference data types the operators compare the memory location values to determine if the objects are the same object or not. These operators return a boolean value of true or false. Example 6.20 shows these operators in action. Figure 6-20 shows the results of running this program.

6.20 EqualityOpTest.java

```

1      public class EqualityOpTest {
2          public static void main(String[] args){
3
4              Object object_1 = new Object();
5              Object object_2 = new Object();
6              Object object_3 = object_2;
7              int int_i = 0;
8              int int_j = 1;
9
10             System.out.println(object_1 == object_2);
11             System.out.println(object_1 != object_2);
12             System.out.println(object_2 == object_3);
13             System.out.println(object_2 != object_3);
14             System.out.println(int_i == int_j);
15             System.out.println(int_i != int_j);
16             System.out.println(5 == 6);
17             System.out.println('r' != 's');
18         }
19     }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/equalityoptest] swodog% java EqualityOpTest
false
true
true
false
false
true
false
true
[Rick-Millers-Computer:~/desktop/equalityoptest] swodog%

```

Figure 6-20: Results of Running Example 6.20

CONDITIONAL AND AND OR OPERATORS &&, ||

The conditional operators perform conditional AND and OR operations. Their operands must be expressions that evaluate to the boolean values true or false. The && (*conditional AND*) operator will return true only if all operand expressions evaluate to true. If the first operand evaluates to false the second operand is not evaluated.

The || (*conditional OR*) operator will return true if any of its operand expressions evaluate to true. If the first operand is true the second operand is not evaluated.

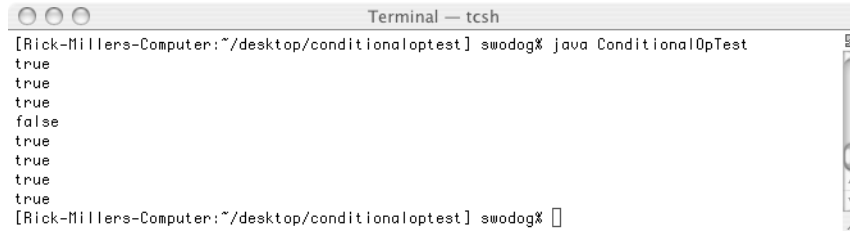
Example 6.21 shows the conditional operators in action. Figure 6-21 shows the results of running this program.

6.21 ConditionalOpTest.java

```

1      public class ConditionalOpTest {
2          public static void main(String[] args){
3              Object object_1 = new Object();
4              Object object_2 = new Object();
5              Object object_3 = object_2;
6              int int_i = 0;
7              int int_j = 1;
8
9              System.out.println((object_1 != null) && (object_2 != null));
10             System.out.println((object_1 != object_2) && (int_i != int_j));
11             System.out.println((object_2 == object_3) || (int_i != int_j));
12             System.out.println((object_2 != object_3) || (5 == 6));
13             System.out.println((int_i == int_j) || (int_i != int_j));
14             System.out.println((int_i != int_j) && ('r' != 's'));
15             System.out.println((5 == 6) || ('r' != 's'));
16             System.out.println(('r' != 's') && ('r' != 't'));
17         }
18     }

```



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/conditionaloptest] swodog% java ConditionalOpTest
true
true
true
false
true
true
true
true
true
[Rick-Millers-Computer:~/desktop/conditionaloptest] swodog% █
```

Figure 6-21: Results of Running Example 6.21

BOOLEAN AND, OR, XOR, AND NOT OPERATORS &, |, ^, !

The boolean operators `&` and `|` perform similar operations to the conditional operators `&&` and `||`. They take boolean operands and return a boolean value of true or false. The difference is that the `&` and `|` operators will evaluate both of their operands regardless of the result of the first operand expression evaluation. These operators are infrequently used.

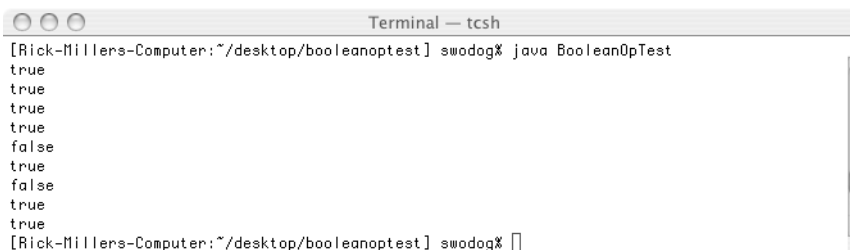
The boolean NOT operator will return the opposite value of the expression to which it is applied. This operator is used extensively.

The boolean XOR (Exclusive OR) will return true if exactly one of its operand expressions evaluates to true.

Example 6.22 shows these operators in action. Figure 6-22 shows the results of running this program.

6.22 *BooleanOpTest.java*

```
1 public class BooleanOpTest {
2     public static void main(String[] args){
3         Object object_1 = new Object();
4         Object object_2 = new Object();
5         Object object_3 = object_2;
6         int int_i = 0;
7         int int_j = 1;
8
9         System.out.println((object_1 != null) & (object_2 != null));
10        System.out.println((object_1 != object_2) | (int_i != int_j));
11        System.out.println((object_2 == object_3) | (int_i != int_j));
12        System.out.println((object_2 == object_3) ^ (!(int_i != int_j)));
13        System.out.println((object_2 != object_3) | (5 == 6));
14        System.out.println((int_i == int_j) | (int_i != int_j));
15        System.out.println((int_i != int_j) ^ ('r' != 's'));
16        System.out.println((5 == 6) | ('r' != 's'));
17        System.out.println(('r' != 's') & ('r' != 't'));
18    }
19 }
```



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/booleanoptest] swodog% java BooleanOpTest
true
true
true
true
false
true
false
true
true
[Rick-Millers-Computer:~/desktop/booleanoptest] swodog% █
```

Figure 6-22: Results of Running Example 6.22

TERNARY CONDITIONAL OPERATOR ?:

It takes a while for novice programmers to warm up to the ternary conditional operator. It is strange to look at but provides some cool functionality. The operator takes three operands. The first is an expression that evaluates to a boolean value of true or false. The second two operands can be of any type so long as they are both the same. The operator returns the value of the second or third operand depending on if the first operand evaluates to true or false. Example 6.23 shows this unique operator in action. This program takes two integers as command-line arguments, converts the Strings to ints, then compares the values to determine if the first argument is less than the second.

6.23 TernaryOpTest.java

```

1 public class TernaryOpTest {
2     public static void main(String[] args){
3         int int_i = Integer.parseInt(args[0]);
4         int int_j = Integer.parseInt(args[1]);
5
6         System.out.println("Is " + int_i + " less than " + int_j + '?');
7         String answer = (int_i < int_j) ? "Yes" : "No";
8         System.out.println(answer);
9
10        int smallest = (int_i < int_j) ? int_i : int_j;
11        System.out.println("Smallest value is: " + smallest);
12    }
13 }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/ternaryoptest] swodog% java TernaryOpTest 6 7
Is 6 less than 7?
Yes
Smallest value is: 6
[Rick-Millers-Computer:~/desktop/ternaryoptest] swodog% java TernaryOpTest 7 6
Is 7 less than 6?
No
Smallest value is: 6
[Rick-Millers-Computer:~/desktop/ternaryoptest] swodog%

```

Figure 6-23: Results of Running Example 6.23

Figure 6-23 shows the program being run twice, first with the values 6 and 7, then with the values 7 and 6. Notice how the ternary operator is used with String data types on line 7 and then with int data types on line 10.

Left Shift and Right Shift Operators <<, >>, >>>

The left and right shift operators take two integer operands. The left-hand operand is the variable whose bits are to be shifted and the right-hand operand is the amount by which the bits are to be shifted.

The left shift operator << will shift a variable's bits to the left while shifting in zeros from the right to replace the shifted bit values.

The right shift operator >> shifts with sign extension, meaning that if the integer variable contains a negative value the sign bit, which is a 1, will be carried in from the left.

The unsigned right shift operator >>> will replace shifted bit values with zeros. Example 6.24 shows the left and right shift operators in action. Figure 6.24 shows the results of running this program.

6.24 ShiftOpTest.java

```

1 public class ShiftOpTest {
2     public static void main(String[] args){
3
4         byte byte_1 = 15;           //00001111
5         int int_1 = 0x0002;        //00000000000000000000000000000010
6
7         System.out.println(byte_1 << 1); // shift left 1 bit
8         System.out.println(byte_1 >> 1); // right shift 2 bits
9         System.out.println(byte_1 >> 2); // right shift 2 bits
10        System.out.println(byte_1 >>> 1); //unsigned right shift 1 bit
11        System.out.println(int_1 << 4); //left shift 4 bits
12        System.out.println(int_1 >>> 6); //unsigned right shift 6 bits
13    }
14 }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/shiftoptest] swodog% java ShiftOpTest
38
7
3
7
32
0
[Rick-Millers-Computer:~/desktop/shiftoptest] swodog%

```

Figure 6-24: Results of Running Example 6.24

INSTANCEOF OPERATOR

The instanceof operator is used to test objects to see if they belong to particular class types. The operator returns a boolean value of true or false depending on the results of the comparison. Example 6.25 shows the instanceof operator in action. Figure 6.25 gives the results of running this program.

6.25 InstanceofOpTest.java

```

1      public class InstanceofOpTest {
2          public static void main(String[] args){
3              Object object_1 = new Object();
4              String string_1 = "I love Java!";
5
6              System.out.println(object_1 instanceof Object);
7              System.out.println(string_1 instanceof String);
8              System.out.println(string_1 instanceof Object);
9              System.out.println(object_1 instanceof String);
10         }
11     }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/instanceofopTest] swodog% java InstanceofOpTest
true
true
true
false
[Rick-Millers-Computer:~/desktop/instanceofopTest] swodog%

```

Figure 6-25: Results of Running Example 6.25

Notice that the use of the instanceof operator on line 8 returns true because class String descends from class Object, therefore a String is an Object. However, an Object is not a String as is shown by the use of the instanceof operator on line 9.

UNARY PREFIX AND POSTFIX INCREMENT AND DECREMENT OPERATORS ++, --

The unary prefix and postfix increment and decrement operators take one operand and increment or decrement it by one. The operands can be any numeric type. The prefix operators are evaluated and the results applied to the current expression. The postfix versions are evaluated after the expression in which they appear. Example 6.26 shows these operators in action. Figure 6.26 gives the results of running this program.

6.26 IncDecOpTest.java

```

1      public class IncDecOpTest {
2          public static void main(String[] args){
3              int int_i = 0;
4              float float_i = 0.0f;
5
6              System.out.println(++int_i); //now it's 1
7              System.out.println(int_i++); //now it's 2, but after 1 is printed
8              System.out.println(--int_i); //2 - 1 = 1
9              System.out.println(int_i--); //1 - 1 = 0, but after 1 is printed
10             System.out.println(int_i); // now print 0
11
12             System.out.println(++float_i); //now it's 1
13             System.out.println(float_i++); //now it's 2, but after 1 is printed
14             System.out.println(--float_i); //2 - 1 = 1
15             System.out.println(float_i--); //1 - 1 = 0, but after 1 is printed
16             System.out.println(float_i); //now print 0
17         }
18     }

```

MEMBER ACCESS OPERATOR .

The member access operator is used to access a class or object's public member fields and methods. An object's fields and methods are accessed via a reference variable while a class's static fields and methods are accessed via the class name. You have seen the member access operator in action in several examples in this and previous chapters. Namely, when a String is converted to an int via the Integer.parseInt() method. You've also seen it used to call the System.out.println() method. You will be introduced to the member access operator in greater detail in chapter 9.

```

Terminal — tcsh
[Rick-Millers-Computer: ~/desktop/incdecoptest] swodog% java IncDecOpTest
1
1
1
1
0
1.0
1.0
1.0
1.0
0.0
[Rick-Millers-Computer: ~/desktop/incdecoptest] swodog%

```

Figure 6-26: Results of Running Example 6.26

BITWISE AND, OR, XOR, AND COMPLEMENT OPERATORS &, |, ^, ~

The bitwise operators perform logical operations on the bits of integer operands according to the truth tables shown in figure 6.27.

AND	
0 & 0	0
0 & 1	0
1 & 0	0
1 & 1	1

OR	
0 0	0
0 1	1
1 0	1
1 1	1

XOR	
0 ^ 0	0
0 ^ 1	1
1 ^ 0	1
1 ^ 1	0

NOT	
0	1
1	0

Figure 6-27: Bitwise Operator Truth Tables

Example 6.27 shows these operators in use. Figure 6.28 shows the results of running this program.

```

1 public class BitwiseOpTest {
2     public static void main(String[] args){
3         int int_i = 0xFFFFFFFF; //11111111111111111111111111111111
4         int int_j = 0;         //00000000000000000000000000000000
5
6         System.out.println(~int_i);
7         System.out.println(~int_j);
8         System.out.println(int_i & int_j);
9         System.out.println(int_i | int_j);
10        System.out.println(int_i ^ int_j);
11    }
12 }

```

6.27 BitwiseOpTest.java

```

Terminal — tcsh
[Rick-Millers-Computer: ~/desktop/BitwiseOpTest] swodog% java BitwiseOpTest
0
-1
0
-1
-1
[Rick-Millers-Computer: ~/desktop/BitwiseOpTest] swodog%

```

Figure 6-28: Results of Running Example 6.27

COMBINATION ASSIGNMENT OPERATORS +=, -=, *=, /=, <<=, >>=, >>>=, &=, |=, ^=

The combination assignment operators combine assignment with the designated operation. These operators provide a shortcut way of writing expressions like the following:

```
operand_1 = operand_1 op operand_2;
```

The operands must be of a type normally compatible with the primary operator. Example 6.28 shows a few of these operators in action. Figure 6.29 shows the results of running this program.

```

1 public class CombinationOpTest {
2     public static void main(String[] args){
3         int int_i = 0;
4         int int_j = 1;

```

6.28 CombinationOpTest.java

```

5         int int_k = 2;
6
7         int_i += int_j;    // same as int_i = int_i + int_j
8         System.out.println(int_i);
9         int_i *= (int_k *= int_k);
10        System.out.println(int_i);
11        int_i /= int_k;
12        System.out.println(int_i);
13    }
14 }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/combinationoptest] swodog% java CombinationOpTest
1
4
1
[Rick-Millers-Computer:~/desktop/combinationoptest] swodog%

```

Figure 6-29: Results of Running Example 6.28

JAVA PRIMITIVE TYPE WRAPPER CLASSES

For every Java primitive data type there exists a corresponding Java wrapper class that can be used to add functionality to primitive types. The wrapper classes are found in the `java.lang` package. Table 6-5 lists the primitive types, their corresponding wrapper classes, and some methods of interest supplied by each wrapper class.

Primitive Type	Wrapper Class	Methods of Interest
char	Character	The Character class provides many class field constants and methods(). There are too many to list them all here. You've seen two Character static class methods in action in this chapter. They are <code>isJavaIdentifierStart(char c)</code> , and <code>isJavaIdentifierPart(char c)</code> . Other methods you may find immediately helpful include: <code>toLowerCase(char c)</code> , <code>toUpperCase(char c)</code> , and <code>toUpperCase(char c)</code> . Take the time to look up this class in the Java platform API documentation.
byte	Byte	<code>parseByte(String s)</code> , <code>toString(byte b)</code> , <code>longValue()</code> , <code>intValue()</code> , <code>shortValue()</code> , <code>doubleValue()</code> , <code>floatValue()</code> , <code>toString()</code>
short	Short	<code>parseShort(String s)</code> , <code>intValue()</code> , <code>longValue()</code> , <code>byteValue()</code> , <code>doubleValue()</code> , <code>intValue()</code> , <code>floatValue()</code> , <code>toString()</code>
long	Long	<code>parseLong(String s)</code> , <code>intValue()</code> , <code>shortValue()</code> , <code>byteValue()</code> , <code>doubleValue()</code> , <code>floatValue()</code> , <code>toString()</code>
int	Integer	<code>parseInt(String s)</code> , <code>longValue()</code> , <code>floatValue()</code> , <code>shortValue()</code> , <code>byteValue()</code> , <code>doubleValue()</code> , <code>toString()</code>
float	Float	<code>parseFloat(String s)</code> , <code>byteValue()</code> , <code>shortValue()</code> , <code>longValue()</code> , <code>intValue()</code> , <code>doubleValue()</code> , <code>toString()</code>
double	Double	<code>parseDouble(String s)</code> , <code>floatValue()</code> , <code>byteValue()</code> , <code>shortValue()</code> , <code>longValue()</code> , <code>intValue()</code> , <code>toString()</code>
boolean	Boolean	<code>getBoolean(String name)</code> , <code>toString(Boolean b)</code> , <code>toString()</code>

Table 6-5: Primitive Type Wrapper Classes

SUMMARY

A Java application is created by adding a `main()` method to an ordinary class definition. The keywords `public` and `class` are used in the class definition. The name of the class is formulated according to the Java identifier naming rules. The body of the class definition appears between the opening and closing braces. Everything in the class body belongs to the class.

The `main()` method appears in the class body. The name of the `main()` method is preceded by the keywords `public`, `static`, and `void`. The `main()` method takes a `String` array as its only parameter. The body of the `main()` method is contained between opening and closing braces.

There are several rules you must follow when formulating identifier names: 1) begin with a valid Java identifier start letter, 2) followed by any number of valid Java identifier part letters or digits, 3) your identifier cannot spell a Java reserved keyword, and 4) your identifier cannot spell the words `true`, `false`, or `null`.

The most difficult part of identifier naming involves deciding on what names to use given the context of your program. The best help you can give yourself in your early programming career is to put some thought into your identifier names. Well-named identifiers will make your source code easy to read. Easy-to-read code is easier to understand. Code that's easy to understand is easy to fix should it contain an error or two. Problems can be quickly spotted when your program reads like a story.

A variable's value can be changed during program runtime. A constant's value must be initialized at the point of declaration and cannot be changed during program runtime.

The Java language supports two distinct type categories: primitive and reference. Arrays are a special case of reference type and can be considered to belong to their own type category. Primitive type variables are assigned their values directly. Reference variables hold the address of an object that is dynamically created in the heap memory with the `new` operator.

The `main()` method can directly access class (static) variables and constants. However, a reference variable is required to access instance variables and constants.

The `main()` method's `String` array can be used to pass command-line `String` arguments to the `main()` method when the program is executed.

The `new` operator is used to allocate system memory for an object and assign the location value to a reference variable. You can reuse a reference variable to hold the address of a new object by simply using the `new` operator to create another object. The Java garbage collector will manage the memory issues automatically.

Objects created with the `new` operator reside in unique memory locations. This memory location may be used by an implementation of the Java virtual machine to generate a unique object identifier value called a hashcode. An object's hashcode value can be discovered by calling an object's `hashCode()` method. (*The `Object.toString()` method calls the `Object.hashCode()` method.*)

Java programs are comprised of statements and expressions. Most statements and expressions contain operators. Operators have precedence, but it is a good idea to use parentheses in a complex expression to ensure it is evaluated as you expect.

All the Java primitive data types have corresponding wrapper classes which provide lots of handy functionality.

Skill-Building Exercises

- 1. Compiling A Simple Class:** Create an application named `MyClass`. Include a `main()` method but leave its body empty. Compile and run the class.
- 2. Add Functionality To MyClass:** Add a statement to the `main()` method of `MyClass` that prints a short text message to the console.
- 3. Add Variables To MyClass:** Add several variables to the `main()` method of `MyClass`. Declare one variable for each type of primitive data type. Initialize them to any value you deem necessary and print their values to the console using the `System.out.println()` method.

4. **Read Input Into Your Program From The Command Line:** Using the `main()` method's `String` parameter write a program that reads five `String`s from the command line and converts them to float types. Declare five float variables to hold the converted values. Use the services of the wrapper class `Float`'s `parseFloat()` method to perform the conversion. Add the five float values together and print their total to the console. (*Use example 6.11 as a guide.*)
5. **Research:** Explore the functionality provided by the primitive type wrapper classes. Pay particular attention to the `Character` class. Write a description of each class constant and method explaining its purpose and function.
6. **Java Primitive Type Value Ranges:** Write a program that prints out the minimum and maximum values that can be stored in the primitive data types. (*Hint: Look to the functionality provided by the wrapper classes.*)
7. **Add Static Variables To MyClass:** Add one or two static class variables to `MyClass`. Access them via the `main()` method and change their values. Print their values to the console after each change. Change their values at least three times.
8. **Add Instance Variables To MyClass:** Add one or more instance variables to `MyClass`. Create an instance of `MyClass` in the `main()` method using the `new` operator. Access each instance variable via the reference variable you created and change their value. Print their values to the console after each value change.
9. **Add Class And Instance Constants To MyClass:** Add one or more class and instance constants to `MyClass` and access them via the `main()` method. Print their values to the console.
10. **Use Different Operators In Programs:** Explore the functionality provided by different Java operators. Use the following Java operators in your `MyClass` program: `-`, `*`, `\`, `&&`, `||`, `<<`, `>>`, `>>>`, `%`, `instanceof`, `!`, `~`. Note the effects of using these operators.

SUGGESTED PROJECTS

1. **Average Five Numbers:** Write a program that takes 5 floating point numbers as input via the command line and computes and prints their average.
2. **Compute The Area:** Write a program that computes the area of a rectangle or square given the input height and width.
3. **Find The Greatest Value:** Write a program that takes two numbers as input and returns the larger of the two. Use the ternary conditional operator to perform the comparison.
4. **Compute Time To Travel:** Write a program that takes as input distance in miles and speed in miles/hour and computes the time it will take to travel that far. The equation required is:

$$t = d/s$$

5. **Compute Average Speed:** Write a program that takes as input the distance traveled and the time it took to travel that distance and compute the average speed it took to cover that distance in that time. The equation required is:

$$s = d/t$$

6. **Compute Fuel Efficiency:** Write a program that takes miles traveled since last fill-up and gallons of gas required to fill your car's tank. Calculate how many miles/gallon your car is getting between fill ups.
7. **Division By Left Shifting:** Write a program that divides an integer by 2 by using the left shift operator. This

project may require more research to fully understand the concepts. Read the integer to be divided from the command line and print the result of the left shift to the console.

SELF-TEST QUESTIONS

1. Label the parts of the following Java program:

```
1      public class QuestionOne {
2          private int int_i;
3          private static int int_j;
4          public static final int INT_K = 25;
5
6          public static void main(String[] args){
7              System.out.println(int_j);
8              System.out.println(INT_K);
9
10             QuestionOne q1 = new QuestionOne();
11
12             System.out.println(q1.int_i);
13         }
14     }
```

2. What is the definition of the term variable?
3. What is the difference between a class and an instance variable?
4. What is the definition of the term constant? How do you declare a constant?
5. List and describe the two Java type categories. Discuss the primary differences between each type category.
6. What are the four requirements for formulating identifier names?
7. What is the function of the new operator?
8. What is the purpose of the Java garbage collector?
9. How do you determine an object's memory location value?
10. What are the primitive type wrapper classes and what functionality do they provide?

REFERENCES

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*, 4th Edition. O'Reilly and Associates, Inc. Sebastopol, CA. ISBN: 0-596-00283-1

James Gosling, et. al. *The Java™ Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, MA. ISBN: 0-201-31008-2

Jon Meyer, et. al. *Java™ Virtual Machine*. O'Reilly and Associates, Inc. Sebastopol, CA. ISBN: 1-56592-194-1

Sun Inc.'s Java 2 Platform API Version 1.4.2 Website: [<http://java.sun.com/j2se/1.4.2/docs/api/index.html>]

NOTES

CHAPTER 7



Key Bridge

CONTROLLING THE FLOW OF PROGRAM EXECUTION

LEARNING OBJECTIVES

- *STATE THE DIFFERENCE BETWEEN SELECTION AND ITERATION STATEMENTS*
- *DESCRIBE THE PURPOSE AND USE OF THE IF STATEMENT*
- *DESCRIBE THE PURPOSE AND USE OF THE IF/ELSE STATEMENT*
- *DESCRIBE THE PURPOSE AND USE OF THE FOR STATEMENT*
- *DESCRIBE THE PURPOSE AND USE OF THE WHILE STATEMENT*
- *DESCRIBE THE PURPOSE AND USE OF THE DO/WHILE STATEMENT*
- *DESCRIBE THE PURPOSE AND USE OF CHAINED IF/ELSE STATEMENTS*
- *DESCRIBE THE PURPOSE AND USE OF NESTED FOR STATEMENTS*
- *DESCRIBE THE PURPOSE AND USE OF THE SWITCH STATEMENT*
- *DESCRIBE THE PURPOSE AND USE OF THE BREAK AND CONTINUE KEYWORDS*
- *DEMONSTRATE YOUR ABILITY TO USE CONTROL-FLOW STATEMENTS IN SIMPLE JAVA PROGRAMS*

INTRODUCTION

Program control-flow statements are an important part of the Java programming language because they allow you to alter the course of program execution while the program is running. The program control-flow statements presented in this chapter fall into two categories: 1) selection statements, and 2) iteration statements.

Selection statements allow you to alter the course of program execution flow based on the result of a conditional expression evaluation. There are three types of selection statements: `if`, `if/else`, and `switch`.

Iteration statements provide a mechanism for repeating one or more program statements based on the result of a conditional expression evaluation. There are three types of iteration statements: `for`, `while`, and `do`. Java version 5 introduced the `for/each` statement for use with arrays and collections. I will therefore postpone coverage of the `for/each` statement until chapter 17.

As you will soon learn, each type of control-flow statement has a unique personality. After reading this chapter you will be able to select and apply the appropriate control-flow statement for the particular type of processing you require, enabling you to write increasingly powerful programs.

In addition to selection and iteration statements I will show you how to use the keywords `break` and `continue`. The proper use of these keywords combined with selection and iteration statements provides a greater level of processing control.

The material you learn in this chapter will fill your Java programming tool bag with lots of powerful tools. You will be pleasantly surprised at what you can program with the aid of program control-flow statements.

SELECTION STATEMENTS

There are three types of Java selection statements: `if`, `if/else`, and `switch`. The use of each of these statements is covered in detail in this section.

if STATEMENT

The `if` statement is used to conditionally execute a block of code based on the results of a conditional expression evaluation. Figure 7-1 graphically shows what happens during the processing of an `if` statement.

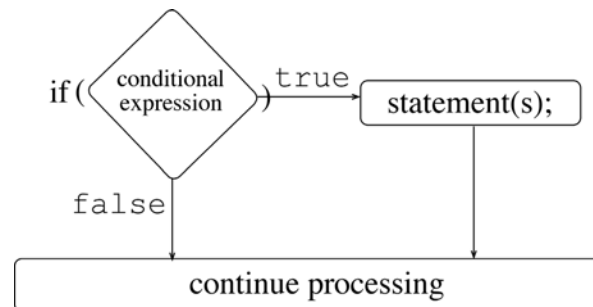


Figure 7-1: `if` Statement Execution Diagram

If the conditional expression contained within the parentheses evaluates to `true` then the body of the `if` statement executes. If the expression evaluates to `false` then the statements contained within the `if` statement body are bypassed and processing continues on to the next statement following the `if` statement.

Example 7.1 shows an `if` statement being used in a simple program that reads two integer values from the command line and compares their values. Figure 7-2 shows the results of running this program.

```

1 public class IfStatementTest {
2     public static void main(String[] args){
3         int int_i = Integer.parseInt(args[0]);
4         int int_j = Integer.parseInt(args[1]);
5
6         if(int_i < int_j)
  
```

7.1 *IfStatementTest.java*

```

7         System.out.println(int_i + " is less than " + int_j);
8     }
9 }

```

```

Terminal — tcsh
java IfStatementTest 4 5
4 is less than 5
[Rick-Millers-Computer:~/desktop/ifstatementtest] swodog% java IfStatementTest 5 4
[Rick-Millers-Computer:~/desktop/ifstatementtest] swodog%

```

Figure 7-2: Results of Running Example 7.1

Referring to example 7.1 — the command-line input is converted from Strings to ints and assigned to the variables `int_i` and `int_j` on lines 3 and 4. The if statement begins on line 6. The less-than operator `<` is used to compare the values of `int_i` and `int_j`. If the value of `int_i` is less than the value of `int_j` then the statement on line 7 is executed. If not, line 7 is skipped and the program exits.

EXECUTING CODE BLOCKS IN IF STATEMENTS

More likely than not you will want to execute multiple statements in the body of an if statement. To do this simply enclose the statements in a set of braces to create a code block. Example 7.2 gives an example of such a code block. Figure 7-3 shows the results of executing this program.

7.2 *IfStatementTest.java (mod 1)*

```

1     public class IfStatementTest {
2         public static void main(String[] args){
3             int int_i = Integer.parseInt(args[0]);
4             int int_j = Integer.parseInt(args[1]);
5
6             if(int_i < int_j) {
7                 System.out.print("Yes ");
8                 System.out.println(int_i + " is less than " + int_j);
9             }
10        }
11    }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/ifstatementtest] swodog% java IfStatementTest 5 6
Yes 5 is less than 6
[Rick-Millers-Computer:~/desktop/ifstatementtest] swodog% java IfStatementTest 121 125
Yes 121 is less than 125
[Rick-Millers-Computer:~/desktop/ifstatementtest] swodog%

```

Figure 7-3: Results of Running Example 7.2

Referring to example 7.2 — notice now that the statements executed by the if statement are contained within a set of braces. The code block begins with the opening brace at the end of line 6 and ends with the closing brace on line 9.

EXECUTING CONSECUTIVE IF STATEMENTS

You can follow one if statement with another if necessary. Example 7.3 gives an example.

7.3 *IfStatementTest.java (mod 2)*

```

1     public class IfStatementTest {
2         public static void main(String[] args){
3             int int_i = Integer.parseInt(args[0]);
4             int int_j = Integer.parseInt(args[1]);
5
6             if(int_i < int_j) {
7                 System.out.print("Yes ");
8                 System.out.println(int_i + " is less than " + int_j);
9             }
10
11            if(int_i >= int_j){
12                System.out.print("No ");
13                System.out.println(int_i + " is not less than " + int_j);

```

```

14     }
15     }
16 }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/ifstatementtest] swodog% java IfStatementTest 5 6
Yes 5 is less than 6
[Rick-Millers-Computer:~/desktop/ifstatementtest] swodog% java IfStatementTest 6 5
No 6 is not less than 5
[Rick-Millers-Computer:~/desktop/ifstatementtest] swodog%

```

Figure 7-4: Results of Running Example 7.3

When example 7.3 is executed the expressions of both if statements are evaluated. When the program is run with the inputs 5 and 6 the expression of the first if statement on line 6 evaluates to true and its body statements are executed. The second if statement's expression evaluates to false and its body statements are skipped.

When the program is run a second time using input values 6 and 5 the opposite happens. This time around the first if statement's expression evaluates to false, its body statements are skipped, and the second if statement's expression evaluates to true and its body statements are executed.

if/ELSE STATEMENT

When you want to provide two possible execution paths for an if statement add the else keyword to form an if/else statement. Figure 7-5 provides an execution diagram of the if/else statement.

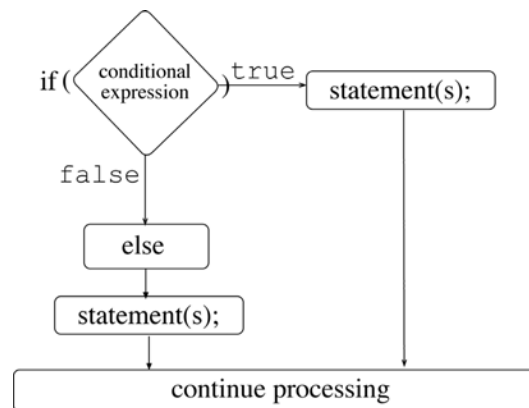


Figure 7-5: if/else Statement Execution Diagram

The if/else statement behaves like the if statement, except now, when the expression evaluates to false, the statements contained within the body of the else clause will be executed. Example 7.4 provides the same functionality as example 7.3 using one if/else statement. Figure 7-6 shows the results of running this program.

```

1     public class IfElseStatementTest {
2         public static void main(String[] args){
3             int int_i = Integer.parseInt(args[0]);
4             int int_j = Integer.parseInt(args[1]);
5
6             if(int_i < int_j) {
7                 System.out.print("Yes ");
8                 System.out.println(int_i + " is less than " + int_j);
9             } else {
10                System.out.print("No ");
11                System.out.println(int_i + " is not less than " + int_j);
12            }
13        }
14    }

```

7.4 IfElseStatementTest.java

```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/ifelsestatementtest] swodog% java IfElseStatementTest 9 10
Yes 9 is less than 10
[Rick-Millers-Computer:~/desktop/ifelsestatementtest] swodog% java IfElseStatementTest 10 9
No 10 is not less than 9
[Rick-Millers-Computer:~/desktop/ifelsestatementtest] swodog% █
```

Figure 7-6: Results of Running Example 7.4

Referring to example 7.4 — the if statement begins on line 6. Should the expression evaluate to true the code block that forms the body of the if statement will execute. Should the expression evaluate to false the code block following the else keyword will execute.

CHAINED if/ELSE STATEMENTS

You can chain if/else statements together to form complex programming logic. To chain one if/else statement to another simply follow the else keyword with an if/else statement. Example 7.5 illustrates the use of chained if/else statements in a program.

7.5 ChainedIfElseTest.java

```
1 public class ChainedIfElseTest {
2     public static void main(String[] args){
3         int int_i = Integer.parseInt(args[0]);
4         int int_j = Integer.parseInt(args[1]);
5
6         if(int_i < int_j) {
7             System.out.print("Yes ");
8             System.out.println(int_i + " is less than " + int_j);
9         } else if(int_i == int_j) {
10            System.out.print("Exact match! ");
11            System.out.println(int_i + " is equal to " + int_j);
12        } else{
13            System.out.print("No ");
14            System.out.println(int_i + " is greater than " + int_j);
15        }
16    }
17 }
```

```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/chainedifelsestatement] swodog% java ChainedIfElseTest 1 2
Yes 1 is less than 2
[Rick-Millers-Computer:~/desktop/chainedifelsestatement] swodog% java ChainedIfElseTest 1 1
Exact match! 1 is equal to 1
[Rick-Millers-Computer:~/desktop/chainedifelsestatement] swodog% java ChainedIfElseTest 2 1
No 2 is greater than 1
[Rick-Millers-Computer:~/desktop/chainedifelsestatement] swodog% █
```

Figure 7-7: Results of Running Example 7.5

There are a couple of important points to note regarding example 7.5. First, notice how the second if/else statement begins on line 9 immediately following the else keyword of the first if/else statement. Second, notice how indenting is used to aid readability.

SWITCH STATEMENT

The switch statement, also referred to as the switch/case statement, is used in situations where you need to provide multiple execution paths based on the evaluation of a particular int, short, byte, or char value. Figure 7-8 gives the execution diagram for a switch statement.

When you write a switch statement you will add one or more case clauses to it. When the switch statement is executed, the value supplied in the parentheses of the switch statement is compared with each case value, and a match results in the execution of that case's code block.

Notice in figure 7-8 how each case's statements are followed by a break statement. The break statement is used to exit the switch and continue processing.

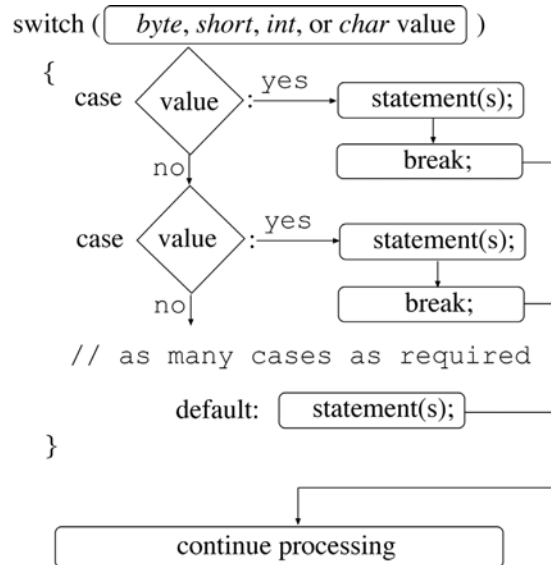


Figure 7-8: switch Statement Execution Diagram

If a break statement is omitted, execution will fall through to the next case, which may or may not be the behavior you require. Example 7.6 gives an example of the switch statement in action. Figure 7-9 shows the results of running this program.

7.6 SwitchStatementTest.java

```

1 public class SwitchStatementTest {
2     public static void main(String[] args){
3         int int_i = Integer.parseInt(args[0]);
4
5         switch(int_i){
6             case 1 : System.out.println("You entered one!");
7                     break;
8             case 2 : System.out.println("You entered two!");
9                     break;
10            case 3 : System.out.println("You entered three!");
11                    break;
12            case 4 : System.out.println("You entered four!");
13                    break;
14            case 5 : System.out.println("You entered five!");
15                    break;
16            default: System.out.println("Please enter a value between 1 and 5");
17        }
18    }
19 }

```

```

Terminal - tcsh
[rick-millers-Computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 1
You entered one
[rick-millers-Computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 2
You entered two
[rick-millers-Computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 3
You entered three
[rick-millers-Computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 4
You entered four
[rick-millers-Computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 5
You entered five
[rick-millers-Computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 6
Please enter a value between 1 and 5
[rick-millers-Computer:~/Desktop/SwitchStatementTest] swodog% █

```

Figure 7-9: Results of Running Example 7.6

Referring to example 7.6 — this program reads a String from the command line and converts it to an integer value using our friend the `Integer.parseInt()` method. The integer variable `int_i` is then used in the switch statement. Its value is compared against the five cases. If there's a match, meaning its value is either 1, 2, 3, 4, or 5, then the related case executes and the appropriate message is printed to the screen. If there's no match then the default case will execute and prompt the user to enter a valid value.

The switch statement shown in example 7.6 can be rewritten to take advantage of case fall-through. It relies on the help of an array which you will see from studying the code shown in example 7.7.

7.7 *SwitchStatementTest.java (mod 1)*

```

1      public class SwitchStatementTest {
2          public static void main(String[] args){
3              int int_i = Integer.parseInt(args[0]);
4              String[] string_array = {"one", "two", "three", "four", "five"};
5
6              switch(int_i){
7                  case 1 :
8                  case 2 :
9                  case 3 :
10                 case 4 :
11                 case 5 : System.out.println("You entered " + string_array[int_i - 1]);
12                     break;
13                 default: System.out.println("Please enter a value between 1 and 5");
14             }
15         }
16     }

```

```

Terminal - tcsh
[rick-millers-computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 1
You entered one
[rick-millers-computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 2
You entered two
[rick-millers-computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 3
You entered three
[rick-millers-computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 4
You entered four
[rick-millers-computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 5
You entered five
[rick-millers-computer:~/Desktop/SwitchStatementTest] swodog% java SwitchStatementTest 6
Please enter a value between 1 and 5
[rick-millers-computer:~/Desktop/SwitchStatementTest] swodog% █

```

Figure 7-10: Results of Running Example 7.7

Referring to example 7.7 — although arrays are formally covered in the chapter 8, the use of one in this particular example should not be too confusing to you. A String array, similar to the one used as an argument to the main() method, is declared on line 4. It is initialized to hold five string values: “one”, “two”, “three”, “four”, and “five”. The switch statement is then rewritten in a more streamlined fashion with the help of the string_array. It is used on line 11, with the help of the variable int_i, to provide the text representation of the numbers 1, 2, 3, 4, and 5. The variable int_i is used to index the array but, as you will notice, 1 must be subtracted from int_i to yield the proper array offset.

If you are confused by the use of the array in this example don't panic. Arrays are covered in painful detail in chapter 8.

NESTED SWITCH STATEMENT

Switch statements can be nested to yield complex programming logic. Study example 7.8.

7.8 *NestedSwitchTest.java*

```

1      public class NestedSwitchTest {
2          public static void main(String[] args){
3              char char_c = args[0].charAt(0);
4              int int_i = Integer.parseInt(args[1]);
5
6              switch(char_c){
7                  case 'U' :
8                  case 'u' : switch(int_i){
9                      case 1:
10                     case 2:
11                     case 3:
12                     case 4:
13                     case 5: System.out.println("You entered " + char_c + " and " + int_i);
14                         break;
15                     default: System.out.println("Please enter: 1, 2, 3, 4, or 5");
16                 }
17                 break;
18                 case 'D' :
19                 case 'd' : switch(int_i){
20                     case 1:
21                     case 2:
22                     case 3:
23                     case 4:

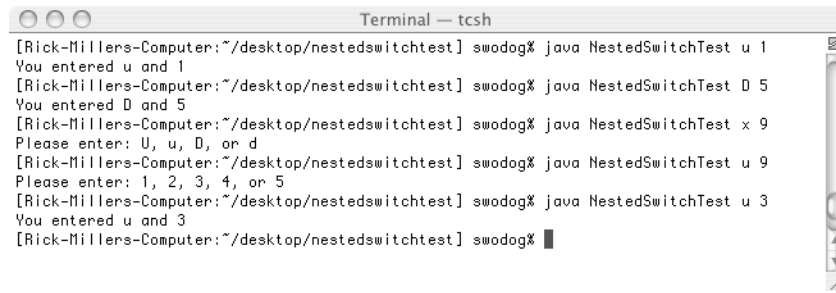
```



```

24         case 5: System.out.println("You entered " + char_c + " and " + int_i);
25             break;
26         default: System.out.println("Please enter: 1, 2, 3, 4, or 5");
27     }
28     break;
29     default: System.out.println("Please enter: U, u, D, or d");
30
31 }
32 }
33 }

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/nestedswitchtest] swodog% java NestedSwitchTest u 1
You entered u and 1
[Rick-Millers-Computer:~/desktop/nestedswitchtest] swodog% java NestedSwitchTest D 5
You entered D and 5
[Rick-Millers-Computer:~/desktop/nestedswitchtest] swodog% java NestedSwitchTest x 9
Please enter: U, u, D, or d
[Rick-Millers-Computer:~/desktop/nestedswitchtest] swodog% java NestedSwitchTest u 9
Please enter: 1, 2, 3, 4, or 5
[Rick-Millers-Computer:~/desktop/nestedswitchtest] swodog% java NestedSwitchTest u 3
You entered u and 3
[Rick-Millers-Computer:~/desktop/nestedswitchtest] swodog% █

```

Figure 7-11: Results of Running Example 7.8

Referring to example 7.8 — this program reads two string arguments from the command line. It then takes the first character of the first string and assigns it to the variable `char_c`. Next, it converts the second string into an int value. The `char_c` variable is used in the first, or outer, switch statement. As you can see from examining the code it is looking for the characters ‘U’, ‘u’, ‘D’, or ‘d’. The nested switch statements are similar to the one used in the previous example. Notice again that indenting is used to help distinguish between outer and inner switch statements.

Quick Review

Selection statements are used to provide alternative paths of program execution. There are three types of selection statements: `if`, `if/else`, and `switch`. The conditional expression of the `if` and `if/else` statements must evaluate to a boolean value of `true` or `false`. Any expression that evaluates to boolean `true` or `false` can be used. You can chain together `if` and `if/else` statements to form complex program logic.

The `switch` statement evaluates a `char`, `byte`, `short`, or `int` value and executes a matching case and its associated statements. Use the `break` keyword to exit a `switch` statement and prevent case fall through. Always provide a default case.

ITERATION STATEMENTS

Iteration statements provide the capability to execute one or more program statements repeatedly based on the results of an expression evaluation. There are three flavors of iteration statement: `while`, `do/while`, and `for`. These statements are discussed in detail in this section. Iteration statements are also referred to as loops. So, when you hear another programmer talking about a `for` loop, `while` loop, or `do` loop, they are referring to the iteration statements.

while STATEMENT

The `while` statement will repeat one or more program statements based on the results of an expression evaluation. Figure 7-12 shows the execution diagram for the `while` statement.

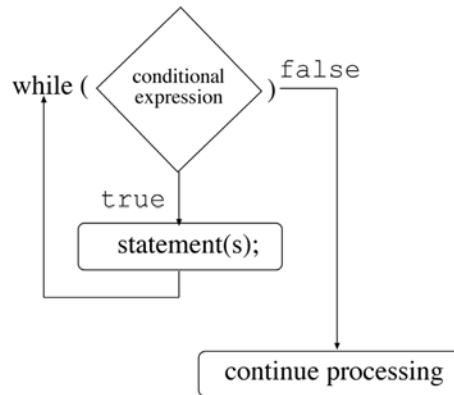


Figure 7-12: while Statement Execution Diagram

PERSONALITY OF THE WHILE STATEMENT

The while statement has the following personality:

- It will evaluate the expression before executing its body statements or code block
- The expression must eventually evaluate to false or the while loop will repeat forever (*Sometimes, however, you want a while loop to repeat forever until some action inside it forces it to exit.*)
- The expression might evaluate to false and therefore not execute its body statements

Essentially, the while loop performs the expression evaluation first, then executes its body statements. Somewhere within the while loop a program statement must either perform an action that will make the expression evaluate to false when the time is right or explicitly force an exit from the while loop.

Example 7.9 shows the while statement in action. Figure 7-13 gives the results of running this program.

7.9 WhileStatementTest.java

```

1     public class WhileStatementTest {
2         public static void main(String[] args){
3             int int_i = 0;
4
5             while(int_i < 10){
6                 System.out.println("The value of int_i = " + int_i);
7                 int_i++;
8             }
9         }
10    }
  
```

```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/whilestatementtest] swodog% java WhileStatementTest
The value of int_i = 0
The value of int_i = 1
The value of int_i = 2
The value of int_i = 3
The value of int_i = 4
The value of int_i = 5
The value of int_i = 6
The value of int_i = 7
The value of int_i = 8
The value of int_i = 9
[Rick-Millers-Computer:~/desktop/whilestatementtest] swodog%
  
```

Figure 7-13: Results of Running Example 7.9

Referring to example 7.9 — on line 3 the integer variable `int_i` is declared and initialized to 0. The variable `int_i` is then used in the while expression and is compared to the integer literal value 10. So long as `int_i` is less than 10 the statements contained within the body of the while loop will execute. (*That includes all statements between the opening brace appearing at the end of line 5 and the closing brace on line 8.*) Notice how the value of `int_i` is incremented with the `++` operator. This is an essential step because if `int_i` is not incremented the expression will always evaluate to true which would result in an infinite loop. (*Note: Apple, Inc. headquarters are located at 1 Infinite Loop, Cupertino, CA 95014*)

do/while STATEMENT

The do/while statement will repeat one or more body statements based on the result of an expression evaluation. The do/while loop is different from the while loop in that its body statements are executed at least once before the expression is evaluated. Figure 7-14 gives the execution diagram for a do/while statement.

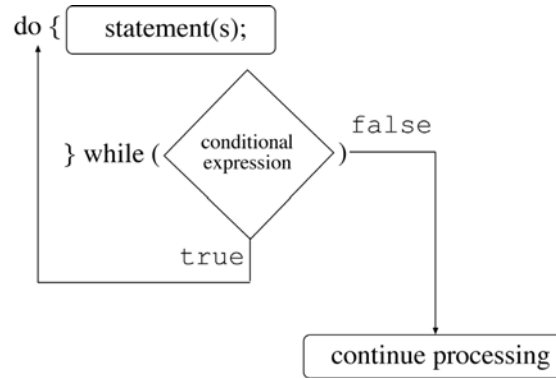


Figure 7-14: do/while Statement Execution Diagram

PERSONALITY OF THE do/while STATEMENT

The do/while statement has the following personality:

- It will execute its body statements once before evaluating its expression
- A statement within its body must take some action that will make the expression evaluate to false or explicitly exit the loop, otherwise the do/while loop will repeat forever (*Like the while loop, you may want it to repeat forever.*)
- Use the do/while loop if the statements it contains must execute at least once

So, the primary difference between the while and do/while statements is where the expression evaluation occurs. The while statement evaluates it at the beginning — the do/while statement evaluates it at the end. Example 7.10 shows the do/while statement in action. Figure 7-15 shows the results of running this program.

7.10 DoWhileStatementTest.java

```

1      public class DoWhileStatementTest {
2          public static void main(String[] args){
3              int int_i = 0;
4
5              do {
6                  System.out.println("The value of int_i = " + int_i);
7                  int_i++;
8              } while(int_i < 10);
9          }
10     }
  
```

```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/dowhilestatementtest] swodog% java DoWhileStatementTest
The value of int_i = 0
The value of int_i = 1
The value of int_i = 2
The value of int_i = 3
The value of int_i = 4
The value of int_i = 5
The value of int_i = 6
The value of int_i = 7
The value of int_i = 8
The value of int_i = 9
[Rick-Millers-Computer:~/desktop/dowhilestatementtest] swodog%
  
```

Figure 7-15: Results of Running Example 7-10

Referring to example 7.10 — on line 3 the integer variable `int_i` is declared and initialized to 0. The do/while statement begins on line 5 and its body statements are contained between the opening brace appearing at the end of line 5 and the closing brace on line 8. Notice that the while keyword and its expression are terminated with a semicolon.

FOR STATEMENT

The for statement provides the capability to repeat a set of program statements just like the while loop and the do/while loop. However, the for loop provides a more convenient way to combine the actions of counter variable initialization, expression evaluation, and counter variable incrementing. Figure 7.16 gives the execution diagram for the for statement.

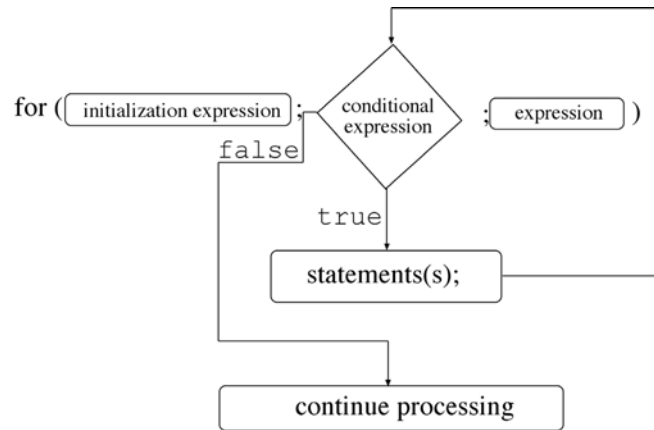


Figure 7-16: for Statement Execution Diagram

HOW THE FOR STATEMENT IS RELATED TO THE WHILE STATEMENT

The for statement is more closely related to the while statement than to the do/while statement. This is because the for statement's middle expression (i.e., *The one used to decide whether or not to repeat its body statements.*) is evaluated before the body statements are executed.

PERSONALITY OF THE FOR STATEMENT

The for statement has the following personality:

- Provides a convenient way to initialize counting variables, perform conditional expression evaluation, and increment loop-control variables
- The conditional expression is evaluated up front just like the while statement
- The for statement is the statement of choice to process arrays (*You will become an expert at using the for statement in chapter 8*)

Example 7.11 shows the for statement in action. Figure 7-17 shows the results of running this program.

7.11 ForStatementTest.java

```

1     public class ForStatementTest {
2         public static void main(String[] args){
3
4             for(int i = 0; i < 10; i++){
5                 System.out.println("The value of i = " + i);
6             }
7         }
8     }
  
```

Referring to example 7.11 — the for statement begins on line 4. Notice how the integer variable *i* is declared and initialized in the first expression. The second expression compares the value of *i* to the integer literal value 10. The third expression increments *i* using the ++ operator.

In this example I have enclosed the single body statement in a code block (*Remember, a code block is denoted by a set of braces.*) However, if a for statement is only going to execute one statement you can omit the braces. (*This goes for the while and do loops as well.*) For example, the for statement shown in example 7.11 could be rewritten in the following manner:

```

for(int i = 0; i < 10; i++)
    System.out.println("The value of i = " + i );
  
```



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/ForStatementTest] swodog% java ForStatementTest
The value of i = 0
The value of i = 1
The value of i = 2
The value of i = 3
The value of i = 4
The value of i = 5
The value of i = 6
The value of i = 7
The value of i = 8
The value of i = 9
[Rick-Millers-Computer:~/desktop/ForStatementTest] swodog% █
```

Figure 7-17: Results of Running Example 7.11

NESTING ITERATION STATEMENTS

Iteration statements can be nested to implement complex programming logic. For instance, you can use a nested for loop to calculate the following summation:

$$\left(\sum_{i=1}^n i \right) \left(\sum_{j=1}^n j \right)$$

Example 7.12 offers one possible solution. Figure 7-18 shows the results of running this program using various inputs.

7.12 NestedForLoop.java

```
1 public class NestedForLoop {
2     public static void main(String[] args){
3         int limit_i = Integer.parseInt(args[0]);
4         int limit_j = Integer.parseInt(args[1]);
5         int total = 0;
6
7         for(int i = 1; i <= limit_i; i++){
8             for(int j = 1; j <= limit_j; j++){
9                 total += (i*j);
10            }
11        }
12        System.out.println("The total is: " + total);
13    }
14 }
```



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/NestedForLoop] swodog% java NestedForLoop 1 1
The total is: 1
[Rick-Millers-Computer:~/desktop/NestedForLoop] swodog% java NestedForLoop 1 2
The total is: 3
[Rick-Millers-Computer:~/desktop/NestedForLoop] swodog% java NestedForLoop 2 1
The total is: 3
[Rick-Millers-Computer:~/desktop/NestedForLoop] swodog% java NestedForLoop 2 2
The total is: 9
[Rick-Millers-Computer:~/desktop/NestedForLoop] swodog% java NestedForLoop 2 3
The total is: 18
[Rick-Millers-Computer:~/desktop/NestedForLoop] swodog% java NestedForLoop 3 3
The total is: 36
[Rick-Millers-Computer:~/desktop/NestedForLoop] swodog% █
```

Figure 7-18: Results of Running Example 7.12

Referring to example 7.12 — this program takes two strings as command-line arguments, converts them into integer values, and assigns them to the variables `limit_i` and `limit_j`. These variables are then used in the middle expression of each for loop to compare against the values of `i` and `j` respectively. Notice too that indenting is used to make it easier to distinguish between the outer and inner for statements.

MIXING SELECTION & ITERATION STATEMENTS: A POWERFUL COMBINATION

You can combine selection and iteration statements in practically any fashion to solve your particular programming problem. You can place selection statements inside the body of iteration statements or vice versa. Refer to the robot rat program developed in chapter 3 for an example of how to combine control-flow statements to form complex

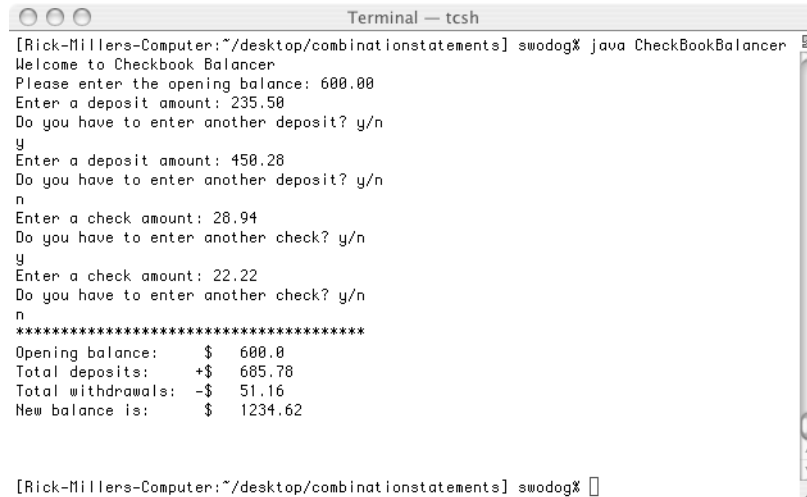
programming logic. Example 7.13 offers another example. The CheckBookBalancer class below implements a simple program that will help you balance your check book. It reads string input from the console and converts it into the appropriate form using the Double primitive-type wrapper class. It also uses try/catch blocks because there is a chance that reading input using the BufferedReader readLine() method may throw an IOException. Figure 7-19 shows an example of CheckBookBalancer in action.

7.13 CheckBookBalancer.java

```

1      import java.io.*;
2
3      public class CheckBookBalancer {
4          public static void main(String[] args){
5              /*** Initialize Program Variables ***/
6              InputStreamReader input_stream = new InputStreamReader(System.in);
7              BufferedReader console = new BufferedReader(input_stream);
8              char keep_going = 'Y';
9              double balance = 0.0;
10             double deposits = 0.0;
11             double withdrawals = 0.0;
12             boolean good_double = false;
13
14             /*** Display Welcome Message ***/
15             System.out.println("Welcome to Checkbook Balancer");
16
17
18             /*** Get Starting Balance ***/
19             do{
20                 try{
21                     System.out.print("Please enter the opening balance: ");
22                     balance = Double.parseDouble(console.readLine());
23                     good_double = true;
24                 }catch(Exception e){ System.out.println("Please enter a valid balance!");}
25                 }while(!good_double);
26
27
28             /*** Add All Deposits ***/
29             while((keep_going == 'y') || (keep_going == 'Y')){
30                 good_double = false;
31                 do{
32                     try{
33                         System.out.print("Enter a deposit amount: ");
34                         deposits += Double.parseDouble(console.readLine());
35                         good_double = true;
36                     }catch(Exception e){ System.out.println("Please enter a valid deposit value!");}
37                 }while(!good_double);
38                 System.out.println("Do you have to enter another deposit? y/n");
39                 try{
40                     keep_going = console.readLine().charAt(0);
41                 }catch(Exception e){ System.out.println("Problem reading input!");}
42             }
43
44             /*** Subtract All Checks Written ***/
45             keep_going = 'Y';
46             while((keep_going == 'y') || (keep_going == 'Y')){
47                 good_double = false;
48                 do{
49                     try{
50                         System.out.print("Enter a check amount: ");
51                         withdrawals += Double.parseDouble(console.readLine());
52                         good_double = true;
53                     }catch(Exception e){ System.out.println("Please enter a valid check amount!");}
54                 }while(!good_double);
55                 System.out.println("Do you have to enter another check? y/n");
56                 try{
57                     keep_going = console.readLine().charAt(0);
58                 }catch(Exception e){ System.out.println("Problem reading input!");}
59             }
60
61             /*** Display Final Tally ***/
62
63             System.out.println("*****");
64             System.out.println("Opening balance:      $ " + balance);
65             System.out.println("Total deposits:      +$ " + deposits);
66             System.out.println("Total withdrawals:  -$ " + withdrawals);
67             balance = balance + (deposits - withdrawals);
68             System.out.println("New balance is:     $ " + balance);
69             System.out.println("\n\n");
70         }
71     }

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/combinati statements] swodog% java CheckBookBalancer
Welcome to Checkbook Balancer
Please enter the opening balance: 600.00
Enter a deposit amount: 235.50
Do you have to enter another deposit? y/n
y
Enter a deposit amount: 450.28
Do you have to enter another deposit? y/n
n
Enter a check amount: 28.94
Do you have to enter another check? y/n
y
Enter a check amount: 22.22
Do you have to enter another check? y/n
n
*****
Opening balance:    $ 600.00
Total deposits:    +$ 685.78
Total withdrawals: -$ 51.16
New balance is:    $ 1234.62

[Rick-Millers-Computer:~/desktop/combinati statements] swodog%

```

Figure 7-19: Results of Running CheckBookBalancer

Quick Review

Iteration statements repeat blocks of program code based on the result of a conditional expression evaluation. There are three types of iteration statements: while, do/while, and for. The while statement evaluates its conditional expression before executing its code block. The do/while statement executes its code block first and then evaluates the conditional expression. The for statement provides a convenient way to write a while statement as it combines loop variable counter declaration and initialization, conditional expression evaluation, and loop counter variable incrementing in a compact format.

BREAK AND CONTINUE

The break and continue statements are used to achieve fine-grained iteration statement control.

The break statement, as you have already learned, is used to exit a switch statement. It is also used to exit for, while, and do loops. There are two forms of break statement: unlabeled and labeled. This section will show you how to use both forms.

The continue statement is used to stop the processing of the current loop iteration and begin the next iteration. It is used in conjunction with for, while, and do loops. It has two forms as well: unlabeled and labeled.

UNLABELED BREAK

An unlabeled break statement looks like this:

```
break;
```

An unlabeled break statement is used in the body of for, while, and do statements to immediately exit the loop and proceed to the next statement. If used within a nested loop structure the unlabeled break statement will only exit its enclosing loop. Example 7.14 shows the unlabeled break statement in action. Figure 7-20 shows the results of running this program.

7.14 BreakStatementTest.java

```

1      public class BreakStatementTest {
2          public static void main(String[] args){
3
4              for(int i = 0; i < 2; i++){
5                  for(int j = 0; j < 1000; j++){

```

```

6         System.out.println("Inner for loop - j = " + j);
7         if(j == 3) break;
8     }
9     System.out.println("Outer for loop - i = " + i);
10    }
11 }
12 }

```

In this example an unlabeled break statement is used to exit a nested for loop. The inner for loop is set to loop 1000 times, however, an if statement on line 7 checks the value of j. If `j == 3` the break statement executes, otherwise the loop is allowed to continue. As soon as `j == 3` evaluates to true the inner for loop is terminated and the outer for loop executes the `System.out.println()` statement on line 9 and then begins another iteration.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/breakstatementtest] swodog% java BreakStatementTest
Inner for loop - j = 0
Inner for loop - j = 1
Inner for loop - j = 2
Inner for loop - j = 3
Outer for loop - i = 0
Inner for loop - j = 0
Inner for loop - j = 1
Inner for loop - j = 2
Inner for loop - j = 3
Outer for loop - i = 1
[Rick-Millers-Computer:~/desktop/breakstatementtest] swodog%

```

Figure 7-20: Results of Running Example 7.14

Labeled BREAK

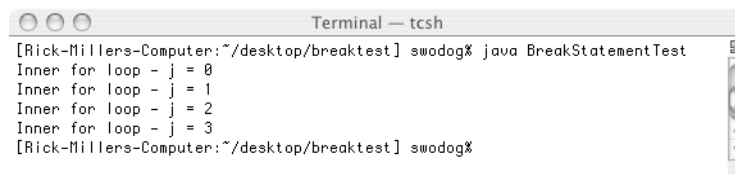
The labeled break statement is kinda cool. It is used to break out of a particular outer or containing loop. To use a labeled break statement you must assign names to the iteration statement you wish to exit and use that name along with the break statement. Example 7.15 shows a labeled break statement in action. Figure 7-21 shows the results of running this program.

7.15 BreakStatementTest.java (mod 1)

```

1     public class BreakStatementTest {
2         public static void main(String[] args){
3
4             Outer_Loop: for(int i = 0; i < 2; i++){
5                 for(int j = 0; j < 1000; j++){
6                     System.out.println("Inner for loop - j = " + j);
7                     if(j == 3) break Outer_Loop;
8                 }
9                 System.out.println("Outer for loop - i = " + i);
10            }
11        }
12    }

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/breaktest] swodog% java BreakStatementTest
Inner for loop - j = 0
Inner for loop - j = 1
Inner for loop - j = 2
Inner for loop - j = 3
[Rick-Millers-Computer:~/desktop/breaktest] swodog%

```

Figure 7-21: Results of Running Example 7.15

Notice in example 7.15 how the outer for loop starting on line 4 is preceded by the label “Outer_Loop:”. Doing so makes the outer for statement a labeled statement. The break statement can now be used on line 7 with the name of the loop it wishes to exit, which in this case is `Outer_Loop`. Notice the difference between the output of this program shown in figure 7-21 and the output of the unlabeled break program shown in figure 7.20.

UNLABELED CONTINUE

The unlabeled continue statement stops the current iteration of its containing loop and begins a new iteration. Example 7.16 shows the unlabeled continue statement in action in a short program that prints odd integers. Figure 7-22 shows this program in action.

7.16 *ContinueStatementTest.java*

```

1      public class ContinueStatementTest {
2          public static void main(String[] args){
3              int limit_i = Integer.parseInt(args[0]);
4
5              for(int i = 0; i < limit_i; i++){
6                  if((i % 2) == 0)
7                      continue;
8                  System.out.println(i);
9              }
10         }
11     }

```

```

Terminal - tcsh
[Rick-Millers-Computer: ~/desktop/ContinueStatementTest] swodog% java ContinueStatementTest 20
1
3
5
7
9
11
13
15
17
19
[Rick-Millers-Computer: ~/desktop/ContinueStatementTest] swodog%

```

Figure 7-22: Results of Running Example 7.16

Referring to example 7.16 — a string argument is entered on the command line, converted into an integer value, and assigned to the `limit_i` variable. The `for` statement uses the `limit_i` variable to determine how many loops it should perform. The `if` statement on line 6 uses the modulus operator `%` to see if the current loop index `i` is evenly divisible by 2. If so, it's not an odd number and the `continue` statement is executed to force another iteration of the `for` loop. If the looping index `i` proves to be odd then the `continue` statement is bypassed and the remainder of the `for` loop executes resulting in the odd number being printed to the console.

LABELED CONTINUE

The labeled continue statement is used to force a new iteration of a labeled iteration statement. It can be used with `for`, `while` and `do` statements just like its unlabeled counterpart. Example 7.17 shows the labeled continue statement in action.

7.17 *ContinueStatementTest.java (mod 1)*

```

1      public class ContinueStatementTest {
2          public static void main(String[] args){
3              int inner_loop_limit = Integer.parseInt(args[0]);
4              int outer_loop_limit = Integer.parseInt(args[1]);
5              int outer_loop_counter = 0;
6              int inner_loop_counter = 0;
7
8              outer_loop: while(outer_loop_counter++ < outer_loop_limit){
9                  while(inner_loop_counter++ < inner_loop_limit){
10                     if((inner_loop_counter % 2) == 0)
11                         continue outer_loop;
12                     System.out.println(inner_loop_counter);
13                 }
14             }
15         }
16     }

```

I admit — example 7.17 is a bit contrived but it does illustrate some good points. First, it calculates and prints odd numbers just like the previous example. It uses nested `while` loops whose loop limits are set via command line arguments. The variables `outer_loop_counter` and `inner_loop_counter` are incremented in the conditional expression

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/ContinueStatementTest] swodog% java ContinueStatementTest 4 4
1
3
[Rick-Millers-Computer:~/desktop/ContinueStatementTest] swodog% java ContinueStatementTest 5 4
1
3
5
[Rick-Millers-Computer:~/desktop/ContinueStatementTest] swodog% java ContinueStatementTest 6 4
1
3
5
[Rick-Millers-Computer:~/desktop/ContinueStatementTest] swodog% java ContinueStatementTest 10 10
1
3
5
7
9
[Rick-Millers-Computer:~/desktop/ContinueStatementTest] swodog%
    
```

Figure 7-23: Results of Running Example 7.17 with Different Loop Limits

of each while statement rather than being incremented in the body of each statement as shown earlier in the chapter. And although it is contrived it is a fun way to experiment with labeled continue statements.

Quick Review

The break and continue keywords provide fine-grained control over iteration statements. In addition to exiting switch statements, the break statement is used to exit for, while, and do loops. There are two forms of break statement: unlabeled and labeled. Unlabeled break exits its immediate containing loop; labeled break is used to exit a labeled iteration statement. The continue keyword terminates the current iteration statement loop and forces the start of a new iteration. Unlabeled continue will force a new iteration of its immediate containing loop; labeled continue will force a new iteration of a labeled iteration statement.

SELECTION AND ITERATION STATEMENT SELECTION TABLE

The following table provides a quick summary of the Java selection and iteration statements. Feel free to copy it and keep it close by your computer until you've mastered their use.

Statement	Execution Diagram	Operation	When To Use
if		Provides an alternative program execution path based on the results of the conditional expression. If conditional expression evaluates to true its body statements are executed. If it evaluates to false they are skipped.	Use the if statement when you need to execute an alternative set of program statements based on some condition.
if/else		Provides two alternative program execution paths based on the results of the conditional expression. If the conditional expression evaluates to true the body of the if statement executes. If it evaluates to false the statements associated with the else clause execute.	Use the if/else when you need to do one thing when the condition is true and another when the condition is false.

Table 7-1: Java Selection And Iteration Statement Selection Guide

Statement	Execution Diagram	Operation	When To Use
switch	<pre> switch (byte, short, int, or char value) { case value : yes statement(s); no break; case value : yes statement(s); no break; // as many cases as required default: statement(s); } continue processing </pre>	<p>The switch statement evaluates char, byte, short, and int values and executes a matching case and its associated statement block. Use the break keyword to exit each case statement to prevent case statement fall-through. Provide a default case.</p>	<p>Use the switch statement in place of chained if/else statements when you are evaluating char, byte, short, or int values.</p>
while	<pre> while (conditional expression) { statement(s); } continue processing </pre>	<p>The while statement repeatedly executes its statement block based on the results of the conditional expression evaluation. The conditional expression will be evaluated first. If true, the statement body will execute and the condition expression will again be evaluated. If it is false the statement body will be skipped and processing will continue as normal.</p>	<p>Use the while loop when you want to do something over and over again while some condition is true.</p>
do/while	<pre> do { statement(s); } while (conditional expression) continue processing </pre>	<p>The do/while statement operates much like the while statement except its body statements will be executed at least once before the conditional expression is evaluated.</p>	<p>Use the do/while statement when you want the body statements to be executed at least once.</p>
for	<pre> for (initialization expression) { statements(s); } continue processing </pre>	<p>The for statement operates like the while statement but offers a more compact way of initializing, comparing, and incrementing counting variables.</p>	<p>Use the for statement when you want to iterate over a set of statements for a known amount of time.</p>

Table 7-1: Java Selection And Iteration Statement Selection Guide

SUMMARY

Program control-flow statements are an important part of the Java programming language because they allow you to alter the course of program execution while the program is running. The program control-flow statements presented in this chapter fall into two general groups: 1) selection statements, and 2) iteration statements.

Selection statements are used to provide alternative paths of program execution. There are three types of selection statements: if, if/else, and switch. The conditional expression of the if and if/else statements must evaluate to a boolean value of true or false. Any expression that evaluates to boolean true or false can be used. You can chain together if and if/else statements to form complex program logic.

The switch statement evaluates a char, byte, short, or int value and executes a matching case and its associated statements. Use the break keyword to exit a switch statement and prevent case fall through. Always provide a default case.

Iteration statements are used to repeat blocks of program code based on the result of a conditional expression evaluation. There are three types of iteration statements: while, do/while, and for. The while statement evaluates its conditional expression before executing its code block. The do/while statement executes its code block first and then evaluates the conditional expression. The for statement provides a convenient way to write a while statement as it combines loop variable counter declaration and initialization, conditional expression evaluation, and loop counter variable incrementing in a compact format.

The break and continue keywords provide finer-grained control over iteration statements. In addition to exiting switch statements, the break statement is used to exit for, while, and do loops. There are two forms of break statement: unlabeled and labeled. Unlabeled break exits its immediate containing loop; labeled break is used to exit a labeled iteration statement. The continue keyword terminates the current iteration statement loop and forces the start of a new iteration. Unlabeled continue will force a new iteration of its immediate containing loop; labeled continue will force a new iteration of a labeled iteration statement.

Skill-Building Exercises

1. **if Statement:** Write a program that reads four string values from the console, converts them into int values using the `Integer.parseInt()` method, and assigns the int values to variables named `int_i`, `int_j`, `int_k`, and `int_l`. Write a series of if statements that perform the conditional expressions shown in the first column of the following table and executes the operations described in the second column. (See *programming examples 7.1 through 7.3* for hints on how to complete this exercise.)

Conditional Expression	Operation
<code>int_i < int_j</code>	Print a text message to the console saying that <code>int_i</code> was less than <code>int_j</code> . Use the values of the variables in the message.
<code>(int_i + int_j) <= int_k</code>	Print a text message to the console showing the values of all the variables and the results of the addition operation.
<code>int_k == int_l</code>	Print a text message to the console saying that <code>int_k</code> was equal to <code>int_l</code> . Use the values of the variables in the text message.
<code>(int_k != int_i) && (int_j > 25)</code>	Print a text message to the console that shows the values of the variables.
<code>(++int_j) & (--int_l)</code>	Print a text message to the console that shows the values of the variables.

Run the program with different sets of input values to see if you can get all the conditional expressions to evaluate to true.

2. **if/else Statement:** Write a program that reads two names from the command line. Assign the names to string variables named `name_1` and `name_2`. Use an if/else statement to compare the text of `name_1` and `name_2` to each other. If the text is equal print a text message to the console showing the names and stating that they are equal. If the text is not equal print a text message to the console showing the names and stating they are not equal. (See *programming example 7.4* for an example of an if/else statement.)

Hint: Use the `String.equals(Object o)` method to perform the string value comparison. For example, given two String objects:

```
String name_1 = "Coralie";
String name_2 = "Coralie";
```

You can compare the text of one String object against the text of another String object by using the `equals()` method in the following fashion:

```
name_1.equals(name_2)
```

The `String.equals(Object o)` method returns a boolean value. If the text of both String objects match it will return true, otherwise it will return false.

3. **switch Statement:** Write a program that reads a string value from the command line and assigns the first character of the string to a char variable named `char_val`. Use a switch statement to check the value of `char_val` and execute a case based on the following table of cases and operations: (See *programming example 7.6* to see a switch statement in action.)

Case	Operation
'A'	Prompt the user to enter two numbers. Add the two numbers the user enters and print the sum to the console. (Hint: Study example 7.13 to see how to process user console input. Remember, the user will enter a string which must be converted to the proper type before performing the add operation.)
'S'	Prompt the user to enter two numbers. Subtract the first number from the second number and print the results to the console.
'M'	Prompt the user to enter two numbers. Multiply them and print the results to the console.
'D'	Prompt the user to enter two numbers. Divide the first number by the second and print the results to the console.
default	Prompt the user to enter two numbers, add them together, and print the sum.

Don't forget to use the `break` keyword to exit each case. Study the `CheckBookBalancer` application shown in example 7.13 for an example of how to process console input.

4. **while Statement:** Write a program that prompts the user to enter a letter. Assign the letter to a char variable named `char_c`. Use a while statement to repeatedly print the following text to the console so long as the user does not enter the letter `'Q'`:

```
"I love Java!"
```

5. **do/while Statement:** Write a program that prompts the user to enter a number. Convert the user's entry into an int and assign the value to an int variable named `int_i`. Use a do/while loop to add the variable to itself five times. Print the results of each iteration of the do/while loop. (Programming example 7.10 shows the do/while statement in action. Hint: You may need to use a separate variable to control the do/while loop.)

6. **for Statement:** Write a program that calculates the following summation using a for statement.

$$\sum_{i=1}^n i^2$$

7. **Chained if/else Statements:** Rewrite skill-building exercise 1 using chained if/else statements.

8. **Mixed Selection and Iteration Statements:** Rewrite skill-building exercise 3. Use a while loop to repeatedly process the switch statement. Exit the while statement if the user enters the character 'E'.

9. **Mixed Selection and Iteration Statements:** Rewrite skill-building exercise 3 again. This time make the while loop execute forever using the following format:

```
while(true) {
    }
}
```

Add a case to the switch statement that exits the program when the user enters the character 'E'. (*Hint: Use the `System.exit()` method.*)

10. **Mixed Selection and Iteration Statements:** Rewrite skill-building exercise 6. Repeatedly prompt the user to enter a value for n, calculate the summation, and print the results of each step of the summation to the console. Keep prompting the user for numbers and perform the operation until they enter a zero.

SUGGESTED PROJECTS

1. **Weight Guesser Game:** Write a program that guesses the user's weight. Have them enter their height and age as a starting point. When the program prints a number it should also ask the user if it is too low, too high, or correct. If correct the program terminates.

2. **Number Guesser Game:** Write a program that generates a random number between 1 and 100 and then asks the user to guess the number. The program must tell the user if their guess is too high or too low. Keep track of the number of user guesses and print the statistics when the user guesses the correct answer. Prompt the user to repeat the game and terminate if they enter 'N'. (*Hint: Use the `java.util.Random` class to generate the random numbers.*)

3. **Simple Calculator:** Write a program that implements a four-function calculator that adds, subtracts, multiplies, and divides.

4. **Calculate Area of Circles:** Write a program that calculates the area of circles using the following formula:

$$\pi r^2$$

Prompt the user for the value of r. After each iteration ask the user if they wish to continue and terminate the program if they enter 'N'. (*Hint: The `java.lang.Math` class defines a public static constant for π .*)

5. **Temperature Converter:** Write a program that converts the temperature from Celsius to Fahrenheit and vice versa. Use the following formulas to perform your conversions:

$$T_f = \left(\left(\frac{9}{5} \right) \times T_c \right) + 32 \qquad T_c = \left(\frac{5}{9} \right) \times (T_f - 32)$$

6. **Continuous Adder:** Write a program that repeatedly adds numbers entered by the user. Display the running total after each iteration. Exit the program when the user enters -9999.
7. **Create Multiplication Tables:** Write a program that prints the multiplication tables for the numbers 1 through 12 up to the 12th factor.
8. **Calculate Hypotenuse:** Write a program that calculates the hypotenuse of triangles using the Pythagorean theorem:

$$a^2 + b^2 = c^2$$

9. **Calculate Grade Averages:** Write a program that helps the instructor calculate test grade averages. Prompt the user for the number of students and then prompt for each grade. Calculate the grade average and print the results.

SELF-TEST QUESTIONS

1. To what type must the conditional expression of a selection or iteration statement evaluate?
2. What's the purpose of a selection statement?
3. What's the purpose of an iteration statement?
4. What four types can be used for switch statement values?
5. What's the primary difference between a while statement and a do/while statement?
6. Explain why, in some programming situations, you would choose to use a do/while statement vs. a while statement.
7. When would you use a switch statement vs. chained if/else statements?
8. For what purpose is the break statement used in switch statements?
9. What's the effect of using an unlabeled break statement in an iteration statement?
10. What's the effect of using a labeled break statement in an iteration statement?
11. What's the effect of using an unlabeled continue statement in an iteration statement?
12. What's the effect of using a labeled continue statement in an iteration statement?

REFERENCES

Lawrence S. Leff. *Geometry The Easy Way*, Second Edition. Barron's Educational Series, Inc. ISBN: 0-8120-4287-5

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*. O'Reilly and Associates, Inc. Sebastopol, CA. ISBN: 0-596-00283-1

NOTES

CHAPTER 8



TIMES SQUARE 2002

ARRAYS

LEARNING OBJECTIVES

- Describe the purpose of an array
- Describe the functionality provided by Java array types
- Trace and describe an array-type inheritance hierarchy
- List and describe the uses of single- and multidimensional arrays
- Describe how Java array objects are allocated in memory
- Describe the difference between arrays of primitive types vs. arrays of reference types
- Use the `java.lang.Object` and `java.lang.Class` classes to discover information about an array
- Demonstrate your ability to create arrays using array literals
- Demonstrate your ability to create arrays using the `new` operator
- Demonstrate your ability to create single-dimensional arrays
- Demonstrate your ability to create multidimensional arrays
- Describe how to access individual array elements via indexing
- Demonstrate your ability to manipulate arrays with iteration statements
- Demonstrate your ability to use the `main()` method's string array parameter

INTRODUCTION

The purpose of this chapter is to give you a solid foundational understanding of arrays and their usage. Since arrays enjoy special status in the Java language you will find them easy to understand and use. This chapter builds upon the material presented in chapters 6 and 7. Here you will learn how to utilize arrays in simple programs and manipulate them with program control-flow statements to yield increasingly powerful programs.

As you will soon learn, arrays are powerful data structures that can be utilized to solve many programming problems. However, even though arrays are powerful, they do have limitations. Detailed knowledge of arrays will give you the ability to judge whether an array is right for your particular application.

In this chapter you will learn the meaning of the term array, how to create and manipulate single- and multidimensional arrays, and how to use arrays in your programs. Starting with single-dimensional arrays of primitive data types, you will learn how to declare array references and how to use the new operator to dynamically create array objects. To help you better understand the concepts of arrays and their use I will show you how they are represented in memory. A solid understanding of the memory concepts behind array allocation will help you to better utilize arrays in your programs. I will then show you how to manipulate single-dimensional arrays using the program control-flow statements you learned in the previous chapter. Understanding the concepts and use of single-dimensional arrays will enable you to easily understand the concepts behind multidimensional arrays.

Along the way you will learn the difference between arrays of primitive types and arrays of reference types. I will show you how to dynamically allocate array element objects and how to call methods on objects via array element references.

Although you will learn a lot about arrays in this chapter, I have omitted some material I feel is best covered later in the book. For instance, I have postponed discussion of how to pass arrays as method arguments until you learn about classes and methods in the next chapter.

WHAT IS AN ARRAY?

An array is a contiguous memory allocation of same-sized or homogeneous data-type elements. The word contiguous means the array elements are located one after the other in memory. The term same-sized means that each array element occupies the same amount of memory space. The size of each array element is determined by the type of objects an array is declared to contain. So, for example, if an array is declared to contain integer primitive types, each element would be the size of an integer and occupy 4-bytes. If, however, an array is declared to contain double primitive types, the size of each element would be 8-bytes. The term homogeneous is often used in place of the term same-sized to refer to objects having the same data type and therefore the same size. Figure 8-1 illustrates these concepts.

This array has 5 elements so it has a length of 5.

Index values range from 0 to (length-1)

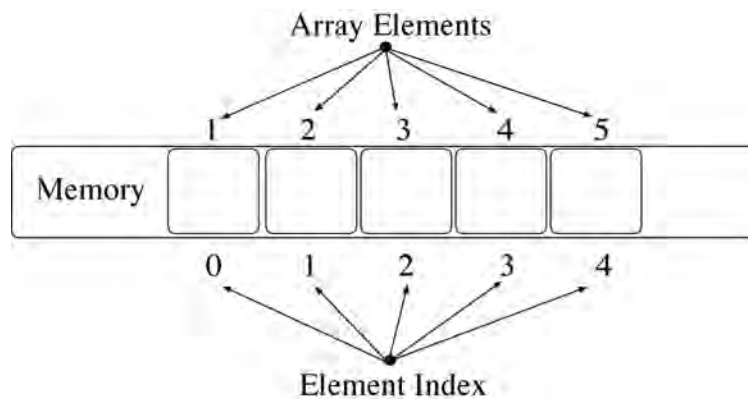


Figure 8-1: Array Elements are Contiguous and Homogeneous

Figure 8-1 shows an array of 5 elements of no particular type. The elements are numbered consecutively beginning with 1 denoting the first element and 5 denoting the last, or 5th, element in the array. Each array element is refer-

enced or accessed by its array index number. Index numbers are always one less than the element number you wish to access. For example, when you want to access the 1st element of an array you will use index number 0. To access the 2nd element of an array you will use index number 1, etc.

The number of elements an array contains is referred to as its length. The array shown in figure 8-1 contains 5 elements so it has a length of 5. The index numbers associated with this array will range from 0 to 4 (*that is 0 to length - 1*).

Specifying Array Types

Array elements can be primitive types, reference types, or arrays of these types. When you declare an array you must specify the type its elements will contain. Figure 8-2 illustrates this concept through the use of the array declaration and allocation syntax.

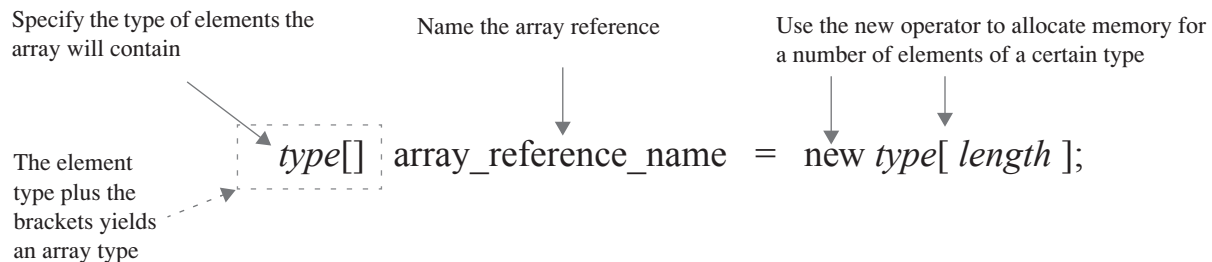


Figure 8-2: Specifying Array Component Type

Figure 8-2 shows the array declaration and allocation syntax for a single-dimensional array having a particular *type* and *length*. The declaration begins with the array element type. The elements of an array can be primitive types or reference types. Reference types can include any reference type (*class or interface*) specified in the Java API or reference types you or someone else create.

The element type is followed by a set of empty brackets. Single-dimensional arrays use one set of brackets. You will add a set of brackets for each additional dimension you want the array to have. (*I cover single- and multidimensional arrays in greater detail below.*) The element type plus the brackets yield an array type. This array type is followed by an identifier that declares the name of the array. To actually allocate memory for an array use the new operator followed by the type of elements the array can contain followed by the length of the array in brackets. The new operator returns a number representing the memory location of the newly created array object and the assignment operator assigns it to the `array_reference_name`.

Figure 8-2 combines the act of declaring an array and the act of creating an array object into one line of code. If required, you could declare an array in one statement and create the array in another. For example, the following line of code declares and allocates memory for a single-dimensional array of integers having length 5:

```
int [] int_array = new int [5];
```

The following line of code would simply declare an array of floats:

```
float [] float_array;
```

And this code would then allocate enough memory to hold 10 float values:

```
float_array = new float [10];
```

The following line of code would declare a two-dimensional array of boolean-type elements and allocate some memory:

```
boolean [] [] boolean_array_2d = new boolean [10] [10];
```

The following line of code would create a single-dimensional array of Strings:

```
String[] string_array = new String[8];
```

You will soon learn the details about single- and multidimensional arrays. If the preceding concepts seem confusing now just hang in there. By the time you complete this chapter you will be using arrays like a pro!

Quick Review

Arrays are contiguously allocated memory elements of homogeneous data types. Contiguous means the elements are arranged in memory one after the other. Homogeneous means each element of the array is of the same data type. An array containing n elements is said to have a length equal to n . Array elements are accessed via their index value which ranges from 0 to length - 1. The index value of a particular array element is always one less than the element number you wish to access. (i.e., 1st element has index 0, 2nd element has index 1, ..., the n^{th} element has index $n-1$)

FUNCTIONALITY PROVIDED BY JAVA ARRAY TYPES

The Java language has two data-type categories: primitive and reference. Array types are a special case of reference types. This means that when you create an array in Java it is an object just like a reference-type object. However, Java arrays possess special features over and above ordinary reference types. These special features make it easy to think of arrays as belonging to a distinct type category all by themselves. This section explains why Java arrays are so special.

ARRAY-TYPE INHERITANCE HIERARCHY

When you declare an array in Java you specify an array type as shown in figure 8-2 above. Just like reference types, array types automatically inherit the functionality of the `java.lang.Object` class. Each array type also implements the `Cloneable` and `Serializable` interfaces. Figure 8-3 gives a UML diagram of an array-type inheritance hierarchy.

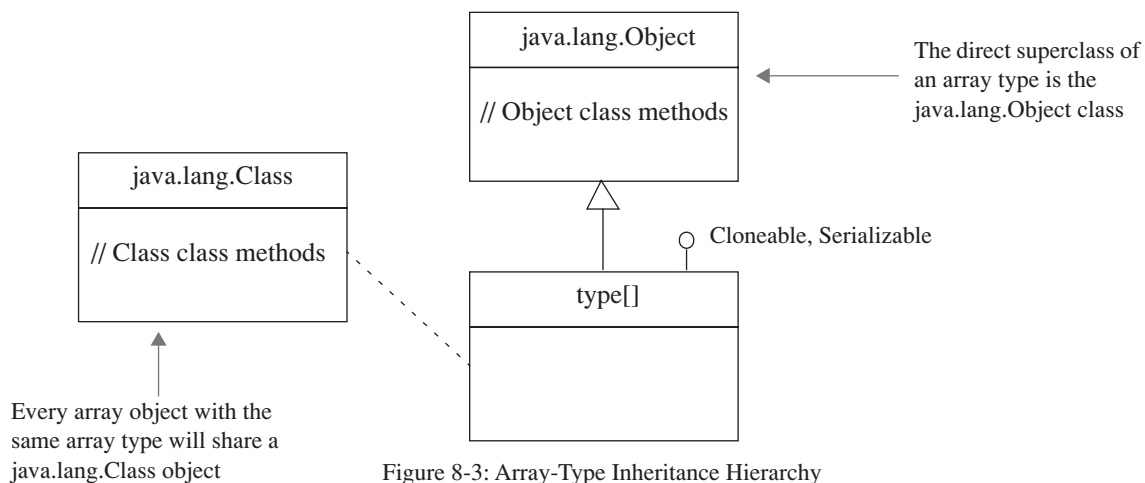


Figure 8-3: Array-Type Inheritance Hierarchy

When an array object is created in memory it will also have a corresponding `java.lang.Class` object. One `Class` object is shared by all array objects having the same array type. You will see how the `Class` and `Object` methods can be used in conjunction with array objects later in the chapter.

THE `java.lang.reflect.Array` CLASS

The `java.lang.reflect.Array` class has a number of static methods that can be used to manipulate array element values. I will not cover the use of the `Array` class in this chapter although I encourage you to explore its functionality on your own by studying its methods and what they do.

SPECIAL PROPERTIES OF JAVA ARRAYS

The following table summarizes the special properties of Java arrays:

Property	Description
Their length cannot be changed once created.	Array objects have an associated length when they are created. The length of an array cannot be changed after the array is created.
Their length can be determined via the <code>length</code> attribute	Array objects have a field named <code>length</code> that contains the value of the length of the array. To access the length attribute use the dot operator and the name of the array. For example: <pre>int [] int_array = new int [5];</pre> This code declares and initializes an array of integer elements with length 5. The next line of code prints the length of the <code>int_array</code> to the console: <pre>System.out.println(int_array.length);</pre>
Array bounds are checked by the Java virtual machine at runtime.	Any attempt to access elements of an array beyond its declared length will result in a runtime exception. This prevents mysterious data corruption bugs that can manifest themselves when misusing arrays in other languages like C or C++.
Array types directly subclass the <code>java.lang.Object</code> class.	Because arrays subclass <code>Object</code> they have the functionality of an <code>Object</code> .
Array types have a corresponding <code>java.lang.Class</code> object.	This means that you can call the <code>java.lang.Class</code> methods on an array object.
Elements are initialized to default values.	Primitive type array elements are initialized to the default value of the particular primitive type each element is declared to contain. Each element of an array of references is initialized to null.

Table 8-1: Java Array Properties

Quick Review

Java array types have special functionality because of their special inheritance hierarchy. Java array types directly subclass the `java.lang.Object` class and implement the `Cloneable` and `Serializable` interfaces. There is also one corresponding `java.lang.Class` object created in memory for each array type in the program.

CREATING AND USING SINGLE-DIMENSIONAL ARRAYS

This section shows you how to declare, create, and use single-dimensional arrays of both primitive and reference types. Once you know how a single-dimensional array works you can easily apply the concepts to multidimensional arrays.

ARRAYS OF PRIMITIVE TYPES

The elements of a primitive type array can be any of the Java primitive types. These include *boolean*, *byte*, *char*, *short*, *int*, *long*, *float*, and *double*. Example 8.1 shows an array of integers being declared, created, and utilized in a short program. Figure 8-4 shows the results of running this program.

8.1 *IntArrayTest.java*

```

1      public class IntArrayTest {
2          public static void main(String[] args){
3              int[] int_array = new int[10];
4              for(int i=0; i<int_array.length; i++){
5                  System.out.print(int_array[i] + " ");
6              }

```

```

7         System.out.println();
8     }
9 }
    
```

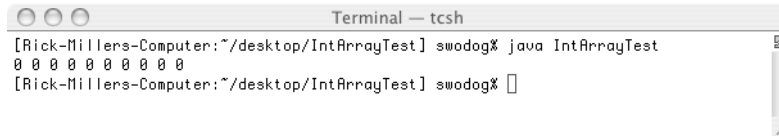


Figure 8-4: Results of Running Example 8.1

Example 8.1 demonstrates several important concepts. First, an array of integer type having length 10 is declared and created on line 3. The name of the array is `int_array`. To demonstrate that each element of the array was automatically initialized to zero, the `for` statement on line 4 iterates over each element of the array beginning with the first element [0] and proceeding to the last element [9], and prints each element value to the console. As you can see from looking at figure 8-4 this results in all zeros being printed to the console.

Notice how each element of `int_array` is accessed via an index value that appears between square brackets appended to the name of the array. (*i.e.*, `int_array[i]`) In this example the value of `i` is controlled by the `for` loop.

HOW PRIMITIVE TYPE ARRAY OBJECTS ARE ARRANGED IN MEMORY

Figure 8-5 shows how the integer primitive type array `int_array` declared and created in example 8.1 is represented in memory.

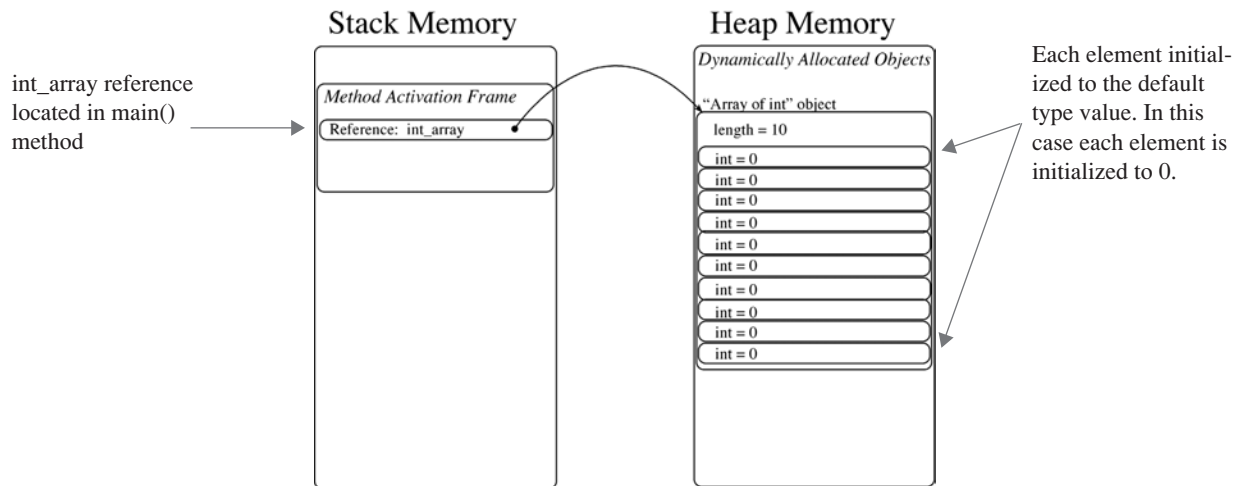


Figure 8-5: Memory Representation of Primitive Type Array `int_array` Showing Default Initialization

The name of the array, `int_array`, is a reference to an object in memory that has type “Array of int”. The array object is dynamically allocated in the heap with the `new` operator and its memory location is assigned to the `int_array` reference. At the time of array object creation each element is initialized to the default value for integers which is 0. The array object’s length attribute is initialized with the value of the length of the array which in this case is 10.

Let’s make a few changes to example 8.1 and assign some values to the `int_array` elements. Example 8.2 adds another `for` loop that initializes each element of `int_array` to the value of the `for` loop index variable `i`. Figure 8-6 shows the results of running this program.

8.2 `IntArrayTest.java (mod 1)`

```

1     public class IntArrayTest {
2         public static void main(String[] args){
3             int[] int_array = new int[10];
4             for(int i=0; i<int_array.length; i++){
5                 System.out.print(int_array[i] + " ");
6             }
7             System.out.println();
8             for(int i=0; i<int_array.length; i++){
9                 int_array[i] = i;
10                System.out.print(int_array[i] + " ");
    
```

```

11     }
12     System.out.println();
13 }
14 }

```

```

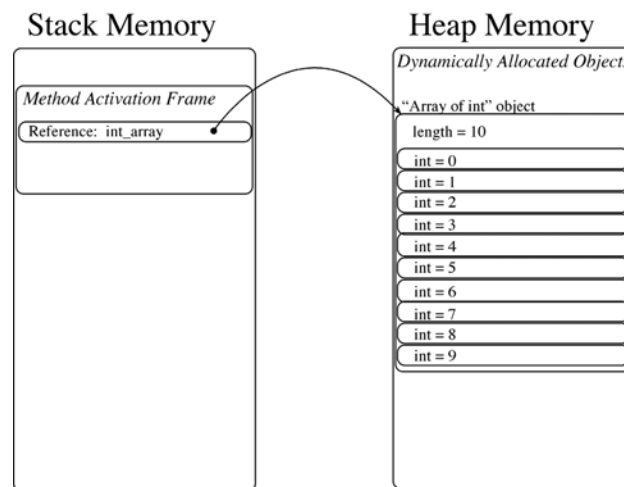
Terminal — tcsh
[Rick-Millers-Computer: ~/desktop/IntArrayTest] swodog% java IntArrayTest
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
[Rick-Millers-Computer: ~/desktop/IntArrayTest] swodog%

```

Figure 8-6: Results of Running Example 8.2

Notice on line 9 of example 8.2 how the value of the second for loop's index variable *i* is assigned directly to each array element. When the array elements are printed to the console you can see that each element's value has changed except for the first which is still zero.

Figure 8-7 shows the memory representation of `int_array` with its new element values.

Figure 8-7: Element Values of `int_array` After Initialization Performed by Second for Loop

Calling Object and Class Methods on Array References

Study the code shown in example 8.3 paying particular attention to lines 15 through 18.

```

1     public class IntArrayTest {
2         public static void main(String[] args){
3             int[] int_array = new int[10];
4             for(int i=0; i<int_array.length; i++){
5                 System.out.print(int_array[i] + " ");
6             }
7             System.out.println();
8             for(int i=0; i<int_array.length; i++){
9                 int_array[i] = i;
10                System.out.print(int_array[i] + " ");
11            }
12            System.out.println();
13
14            /***** Calling Object & Class Methods *****/
15            System.out.println(int_array.getClass().getName());
16            System.out.println(int_array.getClass().getSuperClass());
17            System.out.println(int_array.getClass().isArray());
18            System.out.println(int_array.getClass().isPrimitive());
19        }
20    }

```

8.3 *IntArrayTest.java (mod 2)*

Lines 15 through 18 of example 8.3 above show how methods belonging to Object and Class can be used to get information about an array. The `getClass()` method belongs to `java.lang.Object`. It is used on each line to get the Class


```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/IntArrayTest] swodog% java IntArrayTest
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
[I
class java.lang.Object
true
false
[Rick-Millers-Computer:~/desktop/IntArrayTest] swodog% █

```

Figure 8-8: Results of Running Example 8.3

object associated with the array-type `int[]`. The `getName()`, `getSuperClass()`, `isArray()`, and `isPrimitive()` methods all belong to `java.lang.Class`.

The `getName()` method used on line 15 returns a `String` representing the name of the class. When this method is called it results in the characters “[I” being printed to the console. This represents the class name of an array of integers.

On line 16 the `getSuperClass()` method is called to get the name of `int_array`'s superclass. This results in the string “class java.lang.Object” being printed to the console indicating that the base class of `int_array` is `Object`. On line 17 the `isArray()` method is used to determine if `int_array` is an array. It returns the boolean value `true` as expected. The `isPrimitive()` method is used on line 18 to see if `int_array` is a primitive type. This returns a boolean `false` as expected since although `int_array` is an array of primitive types it is not itself a primitive type.

CREATING SINGLE-DIMENSIONAL ARRAYS USING ARRAY LITERAL VALUES

Up to this point you have seen how memory for an array can be allocated using the `new` operator. Another way to allocate memory for an array and initialize the elements at the same time is to specify the contents of the array using array literal values. The length of the array is determined by the number of literal values appearing in the declaration. Example 8.4 shows two arrays being declared and created using literal values. Figure 8-9 shows the results of running this program.

8.4 ArrayLiterals.java

```

1      public class ArrayLiterals {
2          public static void main(String[] args){
3              int[] int_array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
4              double[] double_array = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
5
6              for(int i = 0; i < int_array.length; i++){
7                  System.out.print(int_array[i] + " ");
8              }
9              System.out.println();
10             System.out.println(int_array.getClass().getName());
11             System.out.println(int_array.getClass().isArray());
12
13             System.out.println();
14
15             for(int i = 0; i < double_array.length; i++){
16                 System.out.print(double_array[i] + " ");
17             }
18             System.out.println();
19             System.out.println(double_array.getClass().getName());
20             System.out.println(double_array.getClass().isArray());
21         }
22     }

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/arrayliterals] swodog% java ArrayLiterals
1 2 3 4 5 6 7 8 9 10
[I
true

1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
[D
true
[Rick-Millers-Computer:~/desktop/arrayliterals] swodog% █

```

Figure 8-9: Results of Running Example 8.4

Example 8.4 declares and initializes two arrays using array literal values. On line 3 an array of integer primitive types named `int_array` is declared. The elements of the array are initialized to the values that appear between the braces. Each element's literal value is separated by a comma. The length of the array is determined by the number of literal values appearing between the braces. The length of `int_array` is 10.

On line 4 an array of double primitive types named `double_array` is declared and initialized with double literal values. The contents of both arrays are printed to the console. `Object` and `Class` methods are then used to determine the characteristics of each array and the results are printed to the console.

DIFFERENCES BETWEEN ARRAYS OF PRIMITIVE TYPES AND ARRAYS OF REFERENCE TYPES

The key difference between arrays of primitive types and arrays of reference types is that primitive type values can be directly assigned to primitive type array elements. The same is not true for reference type elements. In an array of reference types, each element is a reference to an object in memory. When you create an array of references in memory you are not automatically creating each element's object. Each reference element is automatically initialized to null and the object you want it to point to must be explicitly created. (*Or the object must already exist somewhere in memory and be reachable.*) To illustrate these concepts I will use an array of `Objects`. Example 8.5 gives the code for a short program that creates and uses an array of `Objects`. Figure 8-10 shows the results of running this program.

8.5 *ObjectArray.java*

```

1      public class ObjectArray {
2          public static void main(String[] args){
3              Object[] object_array = new Object[10];
4
5              object_array[0] = new Object();
6              System.out.println(object_array[0].getClass());
7              System.out.println(object_array[0].toString());
8              System.out.println();
9
10             object_array[1] = new Object();
11             System.out.println(object_array[1].getClass());
12             System.out.println(object_array[1].toString());
13             System.out.println();
14
15             for(int i = 2; i < object_array.length; i++){
16                 object_array[i] = new Object();
17                 System.out.println(object_array[i].getClass());
18                 System.out.println(object_array[i].toString());
19                 System.out.println();
20             }
21         }
22     }

```

Referring to example 8.5, on line 3 an array of `Objects` of length 10 is declared and created. After line 3 executes the `object_array` reference will point to an array of `Objects` in memory with each element initialized to null as is shown in figure 8-11 below.

On line 5 a new `Object` object is created and its memory location is assigned to the `Object` reference located in `object_array[0]`. The memory picture now looks like figure 8-12 below. Lines 6 and 7 call the `getClass()` and the `toString()` methods on the object pointed to by `object_array[0]`.

The execution of line 10 results in the creation of another `Object` object in memory. The memory picture now looks like figure 8.13. The `for` statement on line 15 creates the remaining `Object` objects and assigns their memory locations to the remaining `object_array` reference elements. Figure 8.14 shows the memory picture after the `for` statement completes execution.

Now that you know the difference between primitive and reference type arrays let's see some single-dimensional arrays being put to good use.

SINGLE-DIMENSIONAL ARRAYS IN ACTION

This section offers several example programs showing how single-dimensional arrays can be used in programs. These programs represent an extremely small sampling of the usefulness arrays afford.



Figure 8-10: Results of Running Example 8.5

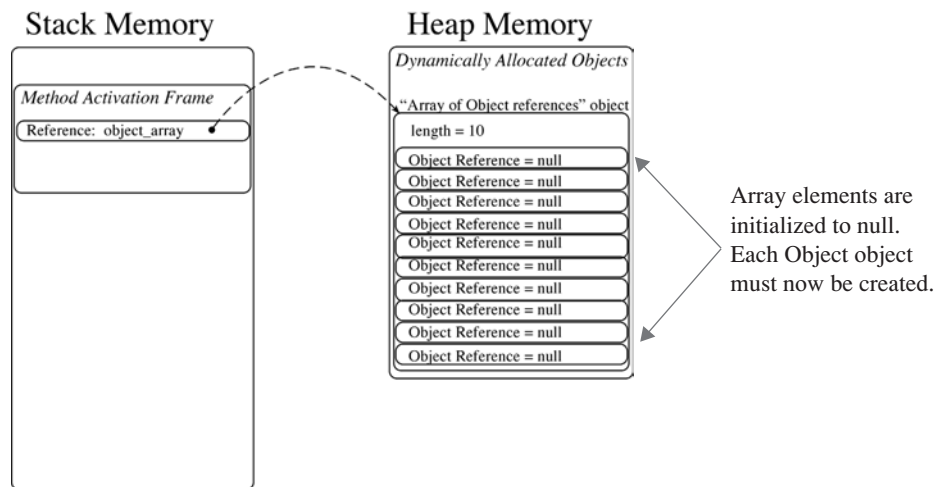


Figure 8-11: State of Affairs After Line 3 of Example 8.5 Executes

MESSAGE ARRAY

One handy use for an array is to store a collection of String messages for later use in a program. Example 8.6 shows how such an array might be utilized. Figure 8-15 shows the results of running this program twice.

8.6 MessageArray.java

```

1      import java.io.*;
2
3      public class MessageArray {
4          public static void main(String[] args){
5              BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
6              String name = null;
7              int int_val = 0;
8
9              String[] messages = {"Welcome to the Message Array Program",
10                                 "Please enter your name: ",
11                                 ", please enter an integer: "};
    
```

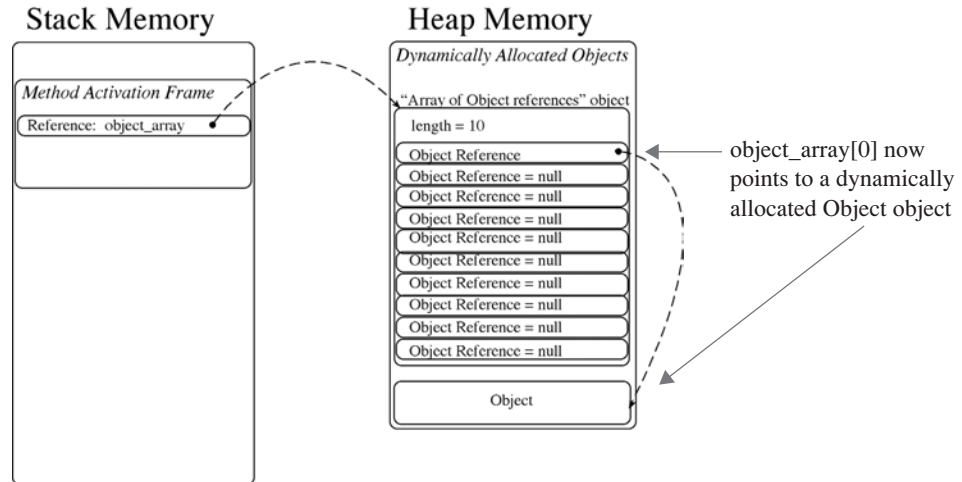


Figure 8-12: State of Affairs After Line 5 of Example 8.5 Executes.

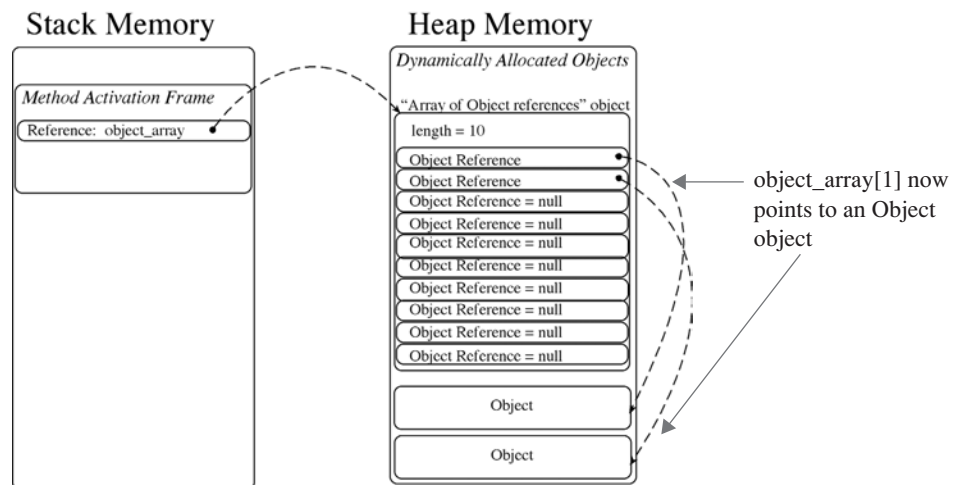


Figure 8-13: State of Affairs After Line 10 of Example 8.5 Executes

```

12         "You did not enter an integer!",
13         "Thank you for running the Message Array program",
14         "Console read error!";
15
16     final int WELCOME_MESSAGE = 0;
17     final int ENTER_NAME_MESSAGE = 1;
18     final int ENTER_INT_MESSAGE = 2;
19     final int INT_ERROR = 3;
20     final int THANK_YOU_MESSAGE = 4;
21     final int CONSOLE_READ_ERROR = 5;
22
23     System.out.println(messages[WELCOME_MESSAGE]);
24     System.out.print(messages[ENTER_NAME_MESSAGE]);
25     try{
26         name = console.readLine();
27     }catch(Exception e){ System.out.println(messages[CONSOLE_READ_ERROR]); }
28
29     System.out.print(name + messages[ENTER_INT_MESSAGE]);
30
31     try{
32         int_val = Integer.parseInt(console.readLine());
33     }catch(NumberFormatException nfe) { System.out.println(messages[INT_ERROR]); }
34     catch(IOException ioe) { System.out.println(messages[CONSOLE_READ_ERROR]); }
35
36     System.out.println(messages[THANK_YOU_MESSAGE]);
37 }
38 }

```

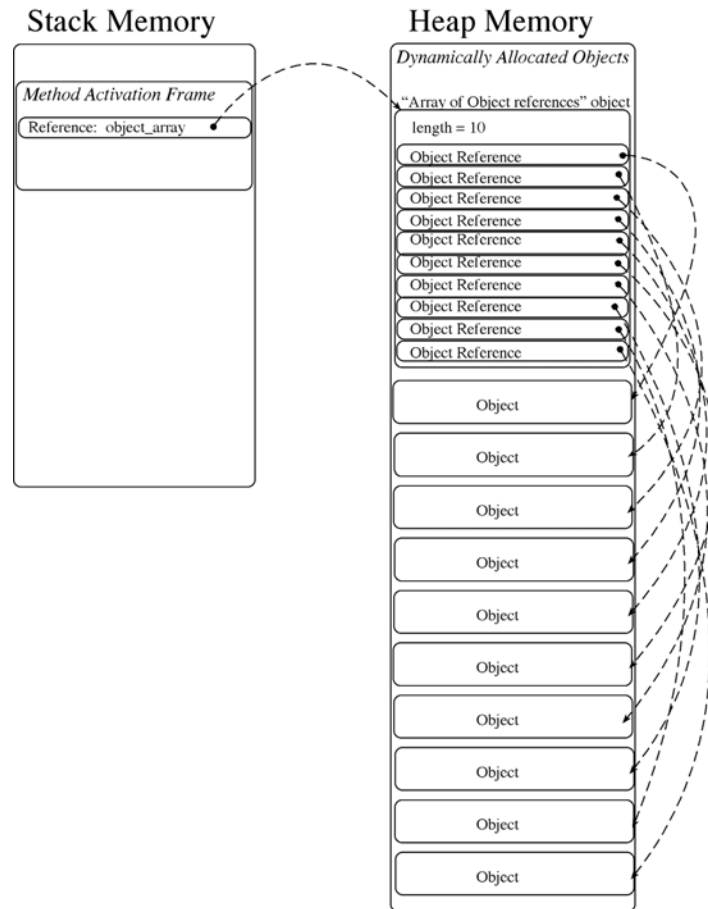


Figure 8-14: Final State of Affairs: All object_array Elements Point to an Object object

```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/MessageArray] swodog% java MessageArray
Welcome to the Message Array Program
Please enter your name: Rick
Rick, please enter an integer: 5
Thank you for running the Message Array program
[Rick-Millers-Computer:~/desktop/MessageArray] swodog% java MessageArray
Welcome to the Message Array Program
Please enter your name: Rick
Rick, please enter an integer: i
You did not enter an integer!
Thank you for running the Message Array program
[Rick-Millers-Computer:~/desktop/MessageArray] swodog% █
```

Figure 8-15: Results of Running Example 8.6

Example 8.6 creates a single-dimensional array of Strings named messages. It initializes each String element using String literals. (*Strings are given special treatment by the Java language and can be initialized using String literal values.*) On lines 16 through 21 an assortment of constants are declared and initialized. These constants are used to index the messages array as is shown on lines 23 and 24.

The program simply asks the user to enter their name followed by a request for them to enter an integer value. Since the `readLine()` method may throw an `IOException` it must be enclosed in a try/catch block. Next, the user is prompted to enter an integer value. Now two things may go wrong. Either, the `readLine()` method may throw an `IOException` or the `Integer.parseInt()` method may throw a `NumberFormatException`. Notice the use of two catch blocks to check for each type of exception. Also notice in figure 8-15 the effects of entering a bad integer value.

Calculating Averages

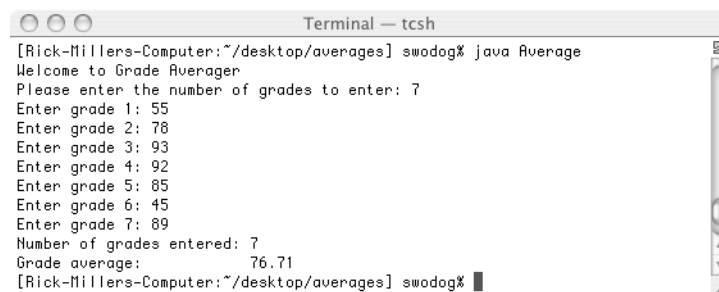
The program given in example 8.7 will calculate class grade averages.

8.7 *Average.java*

```

1      import java.io.*;
2      import java.text.*;
3
4      public class Average {
5          public static void main(String[] args){
6              BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
7              double[] grades      = null;
8              double total         = 0;
9              double average       = 0;
10             int grade_count      = 0;
11             NumberFormat nf = NumberFormat.getInstance();
12
13             nf.setMaximumFractionDigits(2);
14             System.out.println("Welcome to Grade Averager");
15             System.out.print("Please enter the number of grades to enter: ");
16             try{
17                 grade_count = Integer.parseInt(console.readLine());
18             } catch(NumberFormatException nfe) { System.out.println("You did not enter a number!"); }
19             catch(IOException ioe) { System.out.println("Problem reading console!"); }
20
21             if(grade_count > 0){
22                 grades = new double[grade_count];
23                 for(int i = 0; i < grade_count; i++){
24                     System.out.print("Enter grade " + (i+1) + ": ");
25                     try{
26                         grades[i] = Double.parseDouble(console.readLine());
27                     } catch(NumberFormatException nfe) { System.out.println("You did not enter a number!"); }
28                     catch(IOException ioe) { System.out.println("Problem reading console!"); }
29                 } //end for
30
31
32                 for(int i = 0; i < grade_count; i++){
33                     total += grades[i];
34                 } //end for
35
36                 average = total/grade_count;
37                 System.out.println("Number of grades entered: " + grade_count);
38                 System.out.println("Grade average:          " + nf.format(average));
39             } //end if
40         } //end main
41     } // end Average class definition

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/averages] swodog% java Average
Welcome to Grade Averager
Please enter the number of grades to enter: 7
Enter grade 1: 55
Enter grade 2: 78
Enter grade 3: 93
Enter grade 4: 92
Enter grade 5: 85
Enter grade 6: 45
Enter grade 7: 89
Number of grades entered: 7
Grade average:          76.71
[Rick-Millers-Computer:~/desktop/averages] swodog%

```

Figure 8-16: Results of Running Example 8.7

Referring to example 8.7 — an array reference of double primitive types named `grades` is declared on line 7 and initialized to `null`. On lines 8 through 10 several other program variables are declared and initialized. Take special note of what's happening on line 11. Here a `java.text.NumberFormat` reference named `nf` is declared and initialized using the static method `getInstance()` provided by the `NumberFormat` class. Essentially, the `NumberFormat` class provides you with the capability to format numeric output in various ways. The `NumberFormat` reference `nf` is then used on line 13 to set the number of decimal places to display when writing floating point values. I have set the number of decimal places to 2 by calling the `setMaximumFractionDigits()` method with an argument of 2.

The program then prompts the user to enter the number of grades. If this number is greater than 0 then it is used on line 22 to create the `grades` array. The program then enters a `for` loop on line 23, reads each grade from the console, converts it to a double, and assigns it to the i^{th} element of the `grades` array.

After all the grades are entered into the array the grades are summed in the for loop on line 32 and the average is calculated on line 36. Notice how the NumberFormat reference `nf` is used on line 38 to properly format the double value contained in the average variable.

HISTOGRAM: LETTER FREQUENCY COUNTER

Letter frequency counting is an important part of deciphering messages that were encrypted using monalphabetic substitution. Example 8.8 gives the code for a program that counts the occurrences of each letter appearing in a text string and prints a letter frequency display to the console. The program ignores all characters except the 26 letters of the alphabet. Figure 8-17 gives the results of running this program with a sample line of text.

8.8 Histogram.java

```

1      import java.io.*;
2
3      public class Histogram {
4          public static void main(String[] args){
5              BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
6              int letter_frequencies[] = new int[26];
7              final int A = 0, B = 1, C = 2, D = 3, E = 4, F = 5, G = 6,
8                  H = 7, I = 8, J = 9, K = 10, L = 11, M = 12, N = 13,
9                  O = 14, P = 15, Q = 16, R = 17, S = 18, T = 19, U = 20,
10                 V = 21, W = 22, X = 23, Y = 24, Z = 25;
11              String input_string = null;
12
13              System.out.print("Enter a line of characters: ");
14              try {
15                  input_string = console.readLine().toUpperCase();
16                  } catch(IOException ioe) { System.out.println("Problem reading console!"); }
17
18              if(input_string != null){
19                  for(int i = 0; i < input_string.length(); i++){
20                      switch(input_string.charAt(i)){
21                          case 'A': letter_frequencies[A]++;
22                              break;
23                          case 'B': letter_frequencies[B]++;
24                              break;
25                          case 'C': letter_frequencies[C]++;
26                              break;
27                          case 'D': letter_frequencies[D]++;
28                              break;
29                          case 'E': letter_frequencies[E]++;
30                              break;
31                          case 'F': letter_frequencies[F]++;
32                              break;
33                          case 'G': letter_frequencies[G]++;
34                              break;
35                          case 'H': letter_frequencies[H]++;
36                              break;
37                          case 'I': letter_frequencies[I]++;
38                              break;
39                          case 'J': letter_frequencies[J]++;
40                              break;
41                          case 'K': letter_frequencies[K]++;
42                              break;
43                          case 'L': letter_frequencies[L]++;
44                              break;
45                          case 'M': letter_frequencies[M]++;
46                              break;
47                          case 'N': letter_frequencies[N]++;
48                              break;
49                          case 'O': letter_frequencies[O]++;
50                              break;
51                          case 'P': letter_frequencies[P]++;
52                              break;
53                          case 'Q': letter_frequencies[Q]++;
54                              break;
55                          case 'R': letter_frequencies[R]++;
56                              break;
57                          case 'S': letter_frequencies[S]++;
58                              break;
59                          case 'T': letter_frequencies[T]++;
60                              break;
61                          case 'U': letter_frequencies[U]++;
62                              break;
63                          case 'V': letter_frequencies[V]++;
64                              break;

```

```

65         case 'W': letter_frequencies[W]++;
66             break;
67         case 'X': letter_frequencies[X]++;
68             break;
69         case 'Y': letter_frequencies[Y]++;
70             break;
71         case 'Z': letter_frequencies[Z]++;
72             break;
73         default :
74             } //end switch
75     } //end for
76
77     for(int i = 0; i < letter_frequencies.length; i++){
78         System.out.print((char)(i + 65) + ": ");
79         for(int j = 0; j < letter_frequencies[i]; j++){
80             System.out.print('*');
81         } //end for
82         System.out.println();
83     } //end for
84
85     } //end if
86 } // end main
87 } // end Histogram class definition

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/histogram] swodog% java Histogram
Enter a line of characters: Doooooh if you loved Java like I love Java then we'd both love Java too...
A: *****
B: *
C:
D: **
E: *****
F: *
G:
H: ***
I: ***
J: ***
K: *
L: ****
M:
N: *
O: *****
P:
Q:
R:
S:
T: ***
U: *
V: *****
W: *
X:
Y: *
Z:
[Rick-Millers-Computer:~/desktop/histogram] swodog%

```

Figure 8-17: Results of Running Example 8.8

Referring to example 8.8 — on line 6 an integer array named `letter_frequencies` is declared and initialized to contain 26 elements, one for each letter of the alphabet. On lines 7 through 10 several constants are declared and initialized. The constants, named A through Z, are used to index the `letter_frequencies` array later in the program. On line 11 a String reference named `input_string` is declared and initialized to null.

The program then prompts the user to enter a line of characters. The program reads this line of text and converts it all to upper case using the `toUpperCase()` method of the String class. Most of the work is done within the body of the if statement that starts on line 18. If the `input_string` is not null then the for loop will repeatedly execute the switch statement, testing each letter of `input_string` and incrementing the appropriate `letter_frequencies` element.

Take special note on line 19 of how the length of the `input_string` is determined using the String class's `length()` method. A String object is not an array but the methods `length()` and `charAt()` allow a String object to be manipulated on a read-only basis in similar fashion.

Quick Review

Single-dimensional arrays have one dimension — length. You can get an array's length by accessing its length attribute. Arrays can have elements of primitive or reference types. An array type is created by specifying the type

name of array elements followed by one set of brackets []. Use the `java.lang.Object` and `java.lang.Class` methods to get information about an array.

Each element of an array is accessed via an index value contained in a set of brackets. Primitive-type element values can be directly assigned to array elements. When an array of primitive types is created each element is initialized to a default value that's appropriate for the element type. Each element of an array of references is initialized to null. Each object each reference element points to must be dynamically created at some point in the program.

CREATING AND USING MULTIDIMENSIONAL ARRAYS

Java multidimensional arrays are arrays of arrays. For instance, a two-dimensional array is an array of arrays. A three-dimensional array is an array of arrays of arrays, etc. In general, when the elements of an array are themselves arrays, the array is said to be multidimensional. Everything you already know about single-dimensional arrays applies to multidimensional arrays. This means you are already over half way to complete mastery of multidimensional arrays. All you need to know now is how to declare them and use them in your programs.

MULTIDIMENSIONAL ARRAY DECLARATION SYNTAX

You can declare arrays of just about any dimension. However, the most common multidimensional arrays you will see in use are of two or three dimensions. The syntax for declaring and creating a two-dimensional array is shown in figure 8-18.

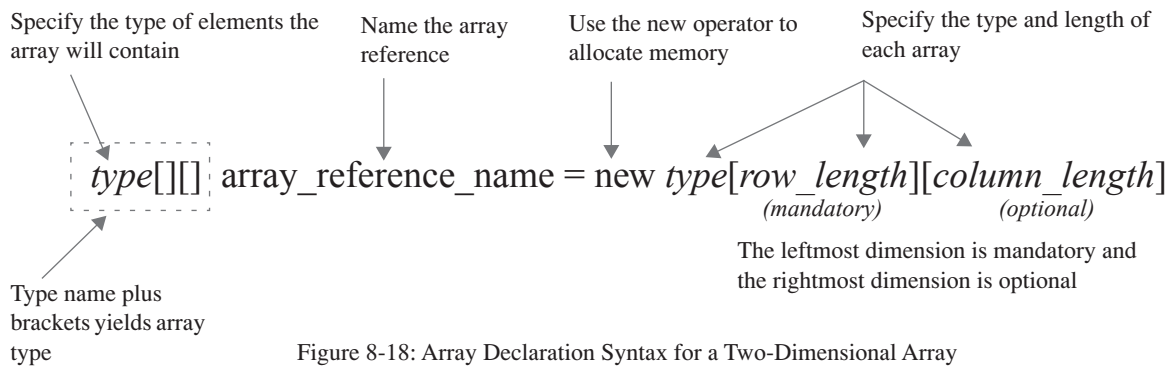


Figure 8-18: Array Declaration Syntax for a Two-Dimensional Array

As figure 8-18 illustrates there are a few new things to keep in mind when declaring multidimensional arrays. First, you will include an extra set of brackets for each additional dimension you want to give your array. Since this is an example of a two-dimensional array there is one extra set of brackets appended to the array type.

When you use the new operator to allocate memory for the array, only the leftmost dimension(s) are mandatory while the rightmost dimension is optional. This enables you to create jagged arrays. (*i.e.*, arrays of different lengths)

Let's take a look at an example of a multidimensional array declaration. The following line of code declares and creates a two-dimensional array of integers with the dimensions 10 by 10:

```
int [] [] int_2d_array = new int [10] [10] ;
```

You could think of this array as being arranged by rows and columns as is shown in figure 8-19.

Each row is a reference to an integer array and there are 10 rows (`int[10] []`) and each column is an element of a 10 integer array (`int[10][10]`). Each element of `int_2d_array` can be accessed using a double set of brackets that indicate the row and column of the desired element. For example, the 3rd element of the 3rd array can be found at `int_2d_array[2][2]`, the 1st element of the 6th array can be found at `int_2d_array[5][0]`, the 6th element of the 6th array can be found at `int_2d_array[5][5]`, and the 9th element of the 10th array can be found at `int_2d_array[9][8]`.

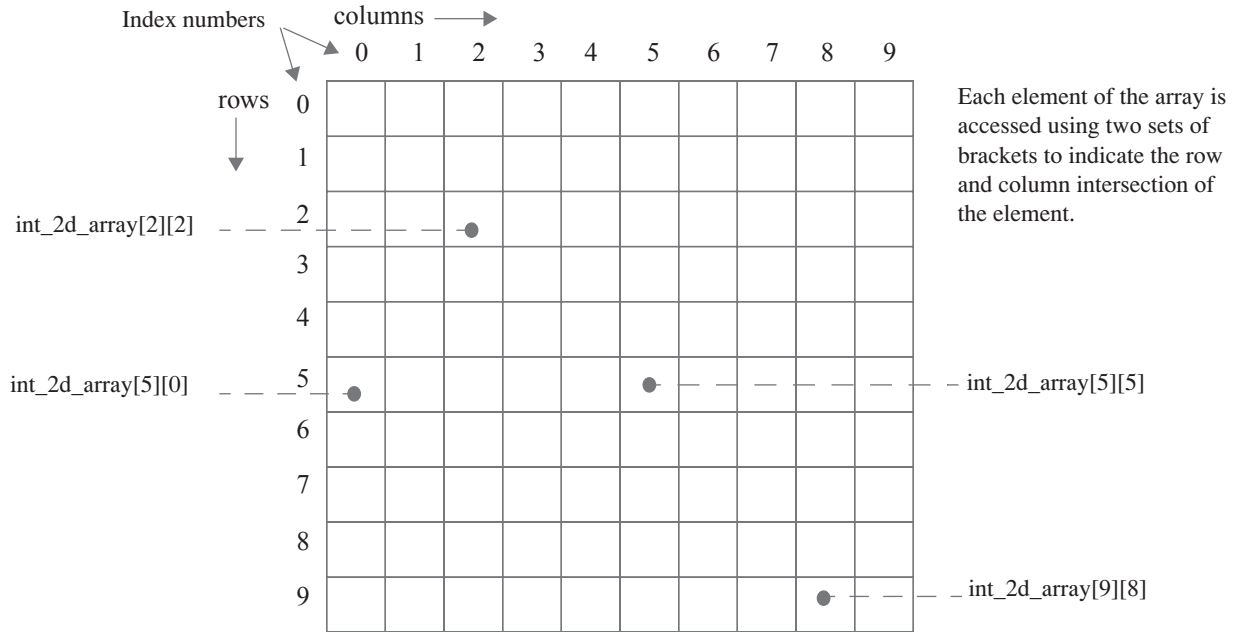


Figure 8-19: A Two Dimensional Array with Dimensions 10 by 10

Example 8.9 is a short program that will let you experiment with creating two dimensional arrays of different dimensions. Figure 8-20 shows this program in action using arrays of various dimensions.

8.9 MultiArrays.java

```

1 public class MultiArrays {
2     public static void main(String[] args){
3         int rows = Integer.parseInt(args[0]);
4         int cols = Integer.parseInt(args[1]);
5
6         int[] [] int_2d_array = new int[rows][cols];
7
8         System.out.println(int_2d_array.getClass());
9
10        for(int i = 0; i<int_2d_array.length; i++){
11            for(int j = 0; j<int_2d_array[i].length; j++){
12                System.out.print(int_2d_array[i][j]);
13            }
14            System.out.println();
15        }
16    }
17 }
    
```

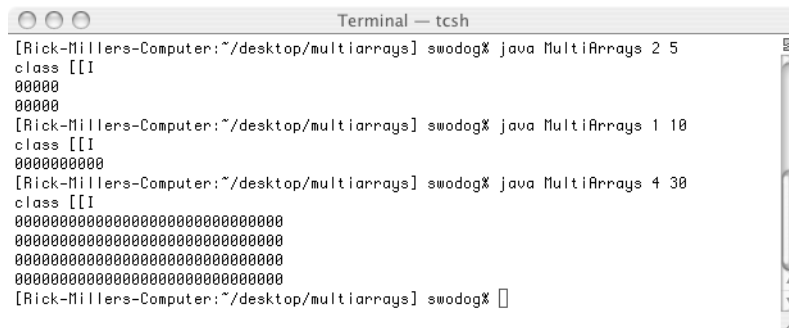


Figure 8-20: Results of Running Example 8.9

Referring to example 8.9 — the `getClass()` method is used on line 8 to determine `int_2d_array`'s class type. As you can see from figure 8-20 its class type is “[[I]” meaning array of int arrays. When this program is run the number of rows desired is entered on the command line followed by the number of columns desired. It simply creates an integer array with the desired dimensions and prints its contents to the console in row order.

MEMORY REPRESENTATION OF A TWO DIMENSIONAL ARRAY

Figure 8-21 offers a memory representation of the `int_2d_array` used in example 8.9 with the dimensions of 2 rows and 10 columns. As you can see, the `int_2d_array` reference declared in the `main()` method points to an array of integer arrays. This array of integer arrays represents the rows. Each “row” element is a reference to an array of integers. The length of each of these integer arrays represents the number of columns contained in the two-dimensional array.

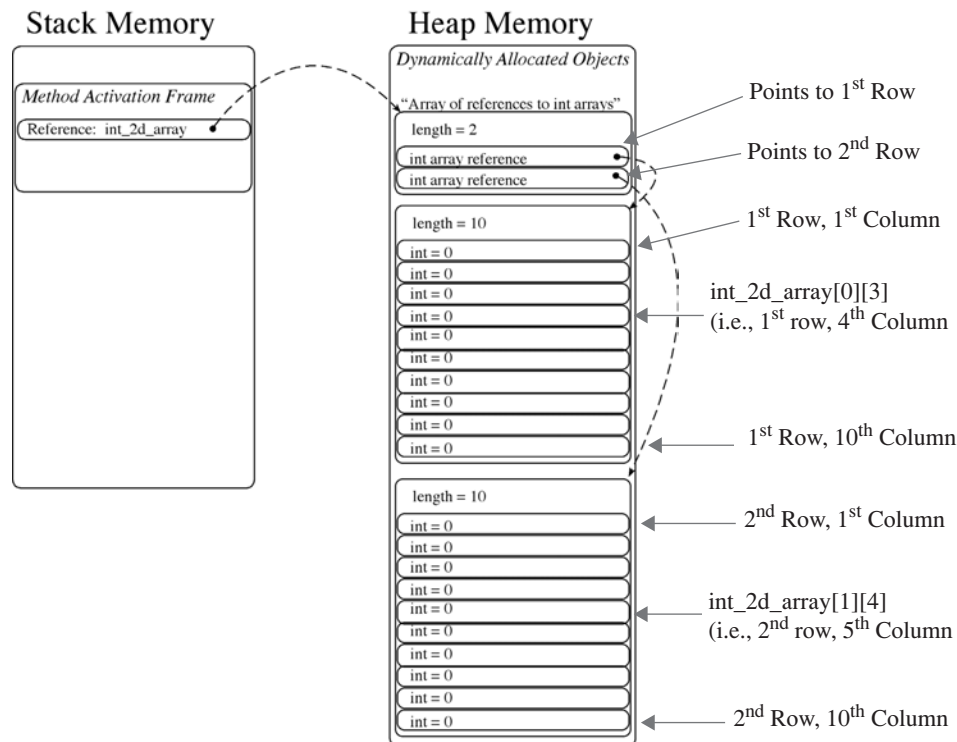


Figure 8-21: Memory Representation of `int_2d_array` with 2 Rows and 10 Columns

CREATING MULTIDIMENSIONAL ARRAYS USING ARRAY LITERALS

Multidimensional arrays can be created using array literals. Example 8.10 is a short program that creates several two-dimensional arrays using array literal values. Figure 8-22 shows the results of running this program.

8.10 TwoDArrayLiterals.java

```

1 public class TwoDArrayLiterals {
2     public static void main(String[] args){
3         /***** ragged int array *****/
4         int[] [] int_2d_ragged_array = {{1,2},
5                                         {1,2,3,4},
6                                         {1,2,3},
7                                         {1,2,3,4,5,6,7,8}};
8
9
10        for(int i = 0; i < int_2d_ragged_array.length; i++){
11            for(int j = 0; j < int_2d_ragged_array[i].length; j++){
12                System.out.print(int_2d_ragged_array[i][j] + " ");
13            }
14            System.out.println();
15        }

```

```

16
17      /***** ragged String array *****/
18      String[][] string_2d_ragged_array = {"Now ", "is ", "the ", "time "},
19                                          {"for ", "all ", "good men "},
20                                          {"to come to ", "the aid "},
21                                          {"of their country!"};
22
23      for(int i = 0; i < string_2d_ragged_array.length; i++){
24          for(int j = 0; j < string_2d_ragged_array[i].length; j++){
25              System.out.print(string_2d_ragged_array[i][j]);
26          }
27          System.out.println();
28      }
29  }
30  }

```

```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/TwoDArrayLiterals] swodog% java TwoDArrayLiterals
1 2
1 2 3 4
1 2 3
1 2 3 4 5 6 7 8
Now is the time
for all good men
to come to the aid
of their country!
[Rick-Millers-Computer:~/desktop/TwoDArrayLiterals] swodog% █

```

Figure 8-22: Results of Running Example 8.10

RAGGED ARRAYS

The arrays created in example 8.10 are known as ragged arrays. A ragged array is a multidimensional array having row lengths of varying sizes. Ragged arrays are easily created using array literals. That's because each row dimension can be explicitly set when the array is declared and created at the same time.

Another way to create a ragged array is to omit the final dimension at the time the array is created, and then dynamically create each array of the final dimension. Study the code shown in example 8.11.

8.11 RaggedArray.java

```

1      import java.io.*;
2
3      public class RaggedArray {
4          public static void main(String[] args){
5              BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
6              int[][] int_2d_ragged_array = new int[5][]; //rightmost dimension omitted
7              int dimension = 0;
8
9              /***** get dimension of each ragged array *****/
10             for(int i = 0; i < int_2d_ragged_array.length; i++){
11                 System.out.print("Enter the ragged array dimension: ");
12                 dimension = 0;
13                 try {
14                     dimension = Integer.parseInt(console.readLine());
15                 }catch(NumberFormatException nfe){ System.out.println("Bad number! Dimension being set to 3");
16                     dimension = 3; }
17                 catch(IOException ioe){ System.out.println("Problem reading console! Dimension being set to 3");
18                     dimension = 3; }
19
20                 int_2d_ragged_array[i] = new int[dimension];
21             }//end for
22
23             /***** print contents of array *****/
24             for(int i = 0; i < int_2d_ragged_array.length; i++){
25                 for(int j = 0; j < int_2d_ragged_array[i].length; j++){
26                     System.out.print(int_2d_ragged_array[i][j]);
27                 } // end inner for
28                 System.out.println();
29             }//end outer for
30
31         }//end main
32     }//end class

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/raggedarray] swodog% java RaggedArray
Enter the ragged array dimension: 2
Enter the ragged array dimension: 6
Enter the ragged array dimension: 9
Enter the ragged array dimension: 4
Enter the ragged array dimension: 3
00
000000
000000000
0000
000
000
[Rick-Millers-Computer:~/desktop/raggedarray] swodog%

```

Figure 8-23: Results of Running Example 8.11

The important point to note about example 8.11 occurs on line 6. Notice how the rightmost array dimension is omitted when the array is created with the new operator. Since the final array dimension is missing, the length of each final array must be explicitly created in the program. This is done in the first for loop that begins on line 10. Each final array is actually created on line 20.

MULTIDIMENSIONAL ARRAYS IN ACTION

The example presented in this section shows how single- and multidimensional arrays can be used together to achieve a high degree of functionality.

Weighted Grade Tool

Example 8.12 gives the code for a class named `WeightedGradeTool`. The program calculates a student's final grade based on weighted grades. Figure 8-24 shows the results of running this program.

8.12 *WeightedGradeTool.java*

```

1      import java.io.*;
2
3      public class WeightedGradeTool {
4          public static void main(String[] args){
5              BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
6              double[][] grades = null;
7              double[] weights = null;
8              String[] students = null;
9              int student_count = 0;
10             int grade_count = 0;
11             double final_grade = 0;
12
13             System.out.println("Welcome to Weighted Grade Tool");
14
15             /***** get student count *****/
16             System.out.print("Please enter the number of students: ");
17             try{
18                 student_count = Integer.parseInt(console.readLine());
19             }catch(NumberFormatException nfe){ System.out.println("That was not an integer!");
20                 System.out.println("Student count will be set to 3.");
21                 student_count = 3; }
22             catch(IOException ioe){ System.out.println("Trouble reading from the console!");
23                 System.out.println("Student count will be set to 3.");
24                 student_count = 3; }
25
26             if(student_count > 0){
27                 students = new String[student_count];
28                 /***** get student names *****/
29                 getNames: for(int i = 0; i < students.length; i++){
30                     System.out.print("Enter student name: ");
31                     try{
32                         students[i] = console.readLine();
33                     } catch(IOException ioe){ System.out.println("Problem reading console!");
34                         System.exit(0); }
35                 } //end getNames for
36
37                 /***** get number of grades per student *****/
38                 System.out.print("Please enter the number of grades to average: ");
39                 try{
40                     grade_count = Integer.parseInt(console.readLine());
41                 }catch(NumberFormatException nfe){ System.out.println("That was not an integer!");

```

```

42         System.out.println("Grade count will be set to 3.");
43         grade_count = 3; }
44     catch(IOException ioe){ System.out.println("Trouble reading from the console!");
45         System.out.println("Grade count will be set to 3.");
46         grade_count = 3; }
47
48     /***** get raw grades *****/
49     grades = new double[student_count][grade_count];
50     getGrades: for(int i = 0; i < grades.length; i++){
51         System.out.println("Enter raw grades for " + students[i]);
52         for(int j = 0; j < grades[i].length; j++){
53             System.out.print("Grade " + (j+1) + ": ");
54             try{
55                 grades[i][j] = Double.parseDouble(console.readLine());
56             }catch(NumberFormatException nfe){ System.out.println("That was not a double!");
57                 System.out.println("Grade will be set to 100");
58                 grades[i][j] = 100; }
59             catch(IOException ioe){ System.out.println("Trouble reading from the console!");
60                 System.out.println("Grade will be set to 100.");
61                 grades[i][j] = 100; }
62
63             }//end inner for
64         }// end getGrades for
65
66     /***** get grade weights *****/
67     weights = new double[grade_count];
68     System.out.println("Enter grade weights. Make sure they total 100%");
69     getWeights: for(int i = 0; i < weights.length; i++){
70         System.out.print("Weight for grade " + (i + 1) + ": ");
71         try{
72             weights[i] = Double.parseDouble(console.readLine());
73         }catch(NumberFormatException nfe){ System.out.println("That was not a double!");
74             System.out.println("Program will exit!");
75             System.exit(0); }
76         catch(IOException ioe){ System.out.println("Trouble reading from the console!");
77             System.out.println("Program will exit!");
78             System.exit(0); }
79
80         }//end getWeights for
81
82     /***** calculate weighted grades *****/
83     calculateGrades: for(int i = 0; i < grades.length; i++){
84         for(int j = 0; j < grades[i].length; j++){
85             grades[i][j] *= weights[j];
86         }//end inner for
87     }//end calculateGrades for
88
89     /***** calculate and print final grade *****/
90     finalGrades: for(int i = 0; i < grades.length; i++){
91         System.out.println("Weighted grades for " + students[i] + ": ");
92         final_grade = 0;
93         for(int j = 0; j < grades[i].length; j++){
94             final_grade += grades[i][j];
95         }//end inner for
96         System.out.println(students[i] +'s final grade is: ' + final_grade );
97     }//end averageGrades for
98
99     }// end if
100 }// end main
    }// end class

```

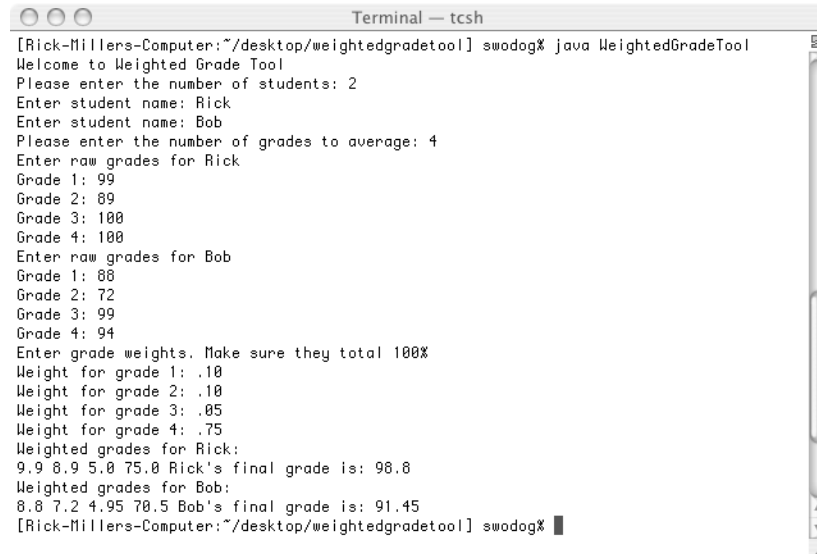
Quick Review

Multidimensional arrays are arrays having more than one dimension. Multidimensional arrays in Java are arrays of arrays. An understanding of single-dimensional Java arrays leads to quick mastery of multidimensional arrays.

To declare a multidimensional array simply add a set of brackets to the array element type for each dimension required in the array. Multidimensional arrays can contain any number of dimensions, however, arrays of two and three dimensions are most common.

Multidimensional arrays can be created using array literal values. The use of array literals enables the initialization of each array element at the point of declaration.

Ragged arrays are multidimensional arrays having final element arrays of various lengths. Ragged arrays can be easily created with array literals. They can also be created by omitting the final dimension at the time of array creation and then dynamically creating each final array with a unique length.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/weightedgradetool] swodog% java WeightedGradeTool
Welcome to Weighted Grade Tool
Please enter the number of students: 2
Enter student name: Rick
Enter student name: Bob
Please enter the number of grades to average: 4
Enter raw grades for Rick
Grade 1: 99
Grade 2: 89
Grade 3: 100
Grade 4: 100
Enter raw grades for Bob
Grade 1: 88
Grade 2: 72
Grade 3: 99
Grade 4: 94
Enter grade weights. Make sure they total 100%
Weight for grade 1: .10
Weight for grade 2: .10
Weight for grade 3: .05
Weight for grade 4: .75
Weighted grades for Rick:
9.9 8.9 5.0 75.0 Rick's final grade is: 98.8
Weighted grades for Bob:
8.8 7.2 4.95 70.5 Bob's final grade is: 91.45
[Rick-Millers-Computer:~/desktop/weightedgradetool] swodog%

```

Figure 8-24: Results of Running Example 8.12

THE MAIN() METHOD'S STRING ARRAY

Now that you have a better understanding of arrays the main() method's String array will make much more sense. This section explains the purpose and use of the main() method's String array.

PURPOSE AND USE OF THE MAIN() METHOD'S STRING ARRAY

The purpose of the main() method's String array is to enable Java applications to accept and act upon command-line arguments. The javac compiler is an example of a program that takes command-line arguments, the most important of which is the name of the file to compile. The previous chapter also gave several examples of accepting program input via the command line. Now that you are armed with a better understanding of how arrays work you now have the knowledge to write programs that accept and process command-line arguments.

Example 8.13 gives a short program that accepts a line of text as a command-line argument and displays it in lower or upper case depending on the first command-line argument. Figure 8.25 shows the results of running this program.

8.13 *CommandLine.java*

```

1      public class CommandLine {
2          public static void main(String[] args){
3              StringBuffer sb = null;
4              boolean upper_case = false;
5
6              /***** check for upper case option *****/
7              if(args.length > 0){
8                  switch(args[0].charAt(0)){
9                      case '-': switch(args[0].charAt(1)){
10                         case 'U' :
11                             case 'u' : upper_case = true;
12                                 break;
13                             default: upper_case = false;
14                         }
15                     break;
16                 default: upper_case = false;
17             }
18             // end outer switch
19
20             sb = new StringBuffer(); //create StringBuffer object
21
22             /***** process text string *****/
23             if(upper_case){

```

```

24         for(int i = 1; i < args.length; i++){
25             sb.append(args[i] + " ");
26         }//end for
27         System.out.println(sb.toString().toUpperCase());
28     }else {
29         for(int i = 0; i < args.length; i++){
30             sb.append(args[i] + " ");
31         }//end for
32         System.out.println(sb.toString().toLowerCase());
33     }//end if/else
34
35     } else { System.out.println("Usage: CommandLine [-U | -u] Text string");}
36
37     }//end main
38 }//end class

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/commandline] swodog% java CommandLine
Usage: CommandLine [-U | -u] Text string
[Rick-Millers-Computer:~/desktop/commandline] swodog% java CommandLine DO YOU LOVE JAVA?
do you love java?
[Rick-Millers-Computer:~/desktop/commandline] swodog% java CommandLine -u i love java!
I LOVE JAVA!
[Rick-Millers-Computer:~/desktop/commandline] swodog% java CommandLine -U i love java so much i could just cry!
I LOVE JAVA SO MUCH I COULD JUST CRY!
[Rick-Millers-Computer:~/desktop/commandline] swodog%

```

Figure 8-25: Results of Running Example 8.13

MANIPULATING ARRAYS WITH THE java.util.ARRAYS CLASS

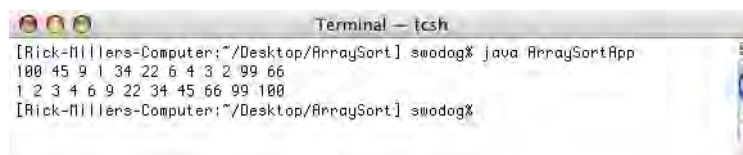
The Java platform makes it easy to perform common array manipulations such as searching and sorting with the `java.util.Arrays` class. Example 8.14 offers a short program that shows the `Arrays` class in action sorting an array of integers. Figure 8-26 shows the results of running this program.

8.14 *ArraySortApp.java*

```

1     import java.util.*;
2
3     public class ArraySortApp {
4         public static void main(String[] args){
5             int[] int_array = {100, 45, 9, 1, 34, 22, 6, 4, 3, 2, 99, 66};
6
7             for(int i=0; i<int_array.length; i++){
8                 System.out.print(int_array[i] + " ");
9             }
10            System.out.println();
11
12            Arrays.sort(int_array);
13
14            for(int i=0; i<int_array.length; i++){
15                System.out.print(int_array[i] + " ");
16            }
17            System.out.println();
18        } // end main() method
19    } // end ArraySortApp class definition

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/Desktop/ArraySort] swodog% java ArraySortApp
100 45 9 1 34 22 6 4 3 2 99 66
1 2 3 4 6 9 22 34 45 66 99 100
[Rick-Millers-Computer:~/Desktop/ArraySort] swodog%

```

Figure 8-26: Results of Running Example 8.14

JAVA API CLASSES USED IN THIS CHAPTER

Table 8-2 lists and describes the Java API classes and interfaces introduced or utilized in this chapter.

Class	Package	Description
Array	java.lang.reflect	The Array class contains methods that allow you to set and query array elements.
Arrays	java.util	The Arrays class provides the capability to perform common array manipulations like searching and sorting on arrays of primitive and reference types.
BufferedReader	java.io	The BufferedReader class applies buffering to character input streams.
Class	java.lang	This class represents a reference type instance in memory. There is one Class object for each array type loaded into the Java virtual machine.
Cloneable	java.lang	The Cloneable interface indicates that the class that implements it may be cloned.
Double	java.lang	The Double class is a wrapper class for the double primitive type.
InputStream	java.io	The InputStream class is an abstract class that is the superclass of all input streams.
InputStreamReader	java.io	The InputStreamReader class is a character input stream.
Integer	java.lang	The Integer class is a wrapper class for the int primitive type.
IOException	java.io	The IOException is the superclass of all input/output exceptions. It signals a problem with an IO operation.
NumberFormat	java.text	The NumberFormat class is used to format numeric output.
NumberFormatException	java.lang	The NumberFormatException signals an illegal number format.
Object	java.lang	The Object class is the root class for all Java and user-defined reference types. This includes Java array types.
Serializable	java.lang	The Serializable interface indicates that the class that implements it may be serialized.
String	java.lang	The String class represents a string of characters and provides methods to manipulate those characters. String objects are immutable, meaning they cannot be changed once created.
StringBuffer	java.lang	The StringBuffer represents a mutable, or changeable, character string.
System	java.lang	The System class provides a platform independent interface to system facilities including properties and input/output streams.

Table 8-2: Java API Classes and Interfaces Referenced in Chapter 8

SUMMARY

Arrays are contiguously allocated memory elements of homogeneous data types. Contiguous means the elements are arranged in memory one after the other. Homogeneous means each element of the array is of the same data type. An array containing n elements is said to have a length equal to n . Array elements are accessed via their index value which ranges from 0 to length - 1. The index value of a particular array element is always one less than the element number you wish to access. (i.e., 1st element has index 0, 2nd element has index 1, ... , the n^{th} element has index $n-1$)

Java array types have special functionality because of their special inheritance hierarchy. Java array types directly subclass the `java.lang.Object` class and implement the `Cloneable` and `Serializable` interfaces. There is also one corresponding `java.lang.Class` object created in memory for each array-type in the program.

Single-dimensional arrays have one dimension — length. You can get an array's length by accessing its `length` attribute. Arrays can have elements of primitive or reference types. An array type is created by specifying the type name of array elements followed by one set of brackets `[]`. Use the `java.lang.Object` and `java.lang.Class` methods to get information about an array.

Each element of an array is accessed via an index value contained in a set of brackets. Primitive-type element values can be directly assigned to array elements. When an array of primitive types is created each element is initialized to a default value that's appropriate for the element type. Each element of an array of references is initialized to null. Each object that each reference element points to must be dynamically created at some point in the program.

Multidimensional arrays are arrays having more than one dimension. Multidimensional arrays in Java are arrays of arrays. An understanding of single-dimensional Java arrays leads to quick mastery of multidimensional arrays.

To declare a multidimensional array simply add a set of brackets to the array element type for each dimension required in the array. Multidimensional arrays can contain any number of dimensions, however, arrays of two and three dimensions are most common.

Multidimensional arrays can be created using array literal values. The use of array literals enables the initialization of each array element at the point of declaration.

Ragged arrays are multidimensional arrays having final element arrays of various lengths. Ragged arrays can be easily created with array literals. They can also be created by omitting the final dimension at the time of array creation and then dynamically creating each final array with a unique length.

Use the `java.util.Arrays` class to perform common manipulations such as searching and sorting on arrays of primitive and reference types.

Skill-Building Exercises

1. **Further Research:** Study the Java API classes and interfaces listed in table 8-2 to better familiarize yourself with the functionality they provide.
2. **Further Research:** Conduct a web search and look for different applications for single- and multidimensional arrays.
3. **Single-Dimensional Arrays:** Write a program that lets you create a single-dimensional array of ints of different sizes at runtime using command line inputs. (*Hint: Refer to example 8.9 for some ideas on how to proceed.*) Print the contents of the arrays to the console.
4. **Single-Dimensional Arrays:** Write a program that reverses the order of text entered on the command line. This will require the use of the `main()` method's String array.
5. **Further Research:** Conduct a web search on different sorting algorithms and how arrays are used to implement these algorithms. Also, several good sources of information regarding sorting algorithms are listed in the references section.

6. **Multidimensional Arrays:** Modify example 8.9 so that it creates two-dimensional arrays of chars. Initialize each element with the character 'c'. Run the program several times to create character arrays of different sizes.
7. **Multidimensional Arrays:** Modify example 8.9 again so that the character array is initialized to the value of the first character read from the command line. (*Hint: Refer to example 8.13 to see how to access the first character of a String.*)
8. **Exploring Arrays Class Functionality:** Research the methods associated with the `java.util.Arrays` class and note their use. Write a short program that creates arrays of floats, ints, doubles, and Strings and use the various sort methods found in the `Arrays` class to sort each array. Print the contents of each array to the console both before and after sorting.

SUGGESTED PROJECTS

1. **Matrix Multiplication:** Given two matrices A_{ij} and B_{jk} the product C_{ik} can be calculated with the following equation:

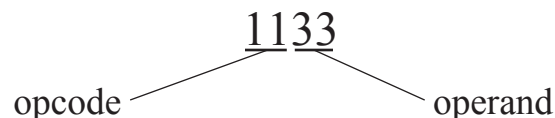
$$C_{ik} = \sum_{j=1}^n A_{ij} B_{jk}$$

Write a program that multiplies the following matrices together and stores the results in a new matrix. Print the resulting matrix values to the console.

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

2. **Modify Histogram Program:** Modify the Histogram program given in example 8.8 so that it can count the occurrence of the digits 0 through 9 and the punctuation marks period '.', comma ',', question mark '?', colon ':', and semi-colon ';'.
3. **Computer Simulator:** You are a Java developer with a high-tech firm doing contract work for the Department of Defense. Your company has won the proposal to develop a proof-of-concept model for an Encrypted Instruction Set Computer System Mark 1. (EISCS Mk1) Your job is to simulate the operation of the EISCS Mk1 with a Java application.

Supporting Information: The only language a computer understands is its machine language instruction set. The EISCS Mk1 is no different. The EISCS machine language instruction set will consist of a four-digit integer with the two most significant digits being the *operation code (opcode)* and the two least significant digits being the *operand*. For example, in the following instruction...



...the number 11 represents the opcode and the number 33 represents the operand. The following table lists and describes each EISCS machine instruction.

Opcode	Mnemonic	Description
<i>Input/Output Operations</i>		
10	READ	Reads an integer value from the console and stores it in memory location identified by the <i>operand</i> .
11	WRITE	Writes the integer value stored in memory location <i>operand</i> to the console.
<i>Load/Store Operations</i>		
20	LOAD	Loads the integer value stored at memory location <i>operand</i> into the accumulator.
21	STORE	Stores the integer value residing in the accumulator into memory location <i>operand</i> .
<i>Arithmetic Operations</i>		
30	ADD	Adds the integer value located in memory location <i>operand</i> to the value stored in the accumulator and leaves the result in the accumulator.
31	SUB	Subtracts the integer value located in memory location <i>operand</i> from the value stored in the accumulator and leaves the result in the accumulator.
32	MUL	Multiplies the integer value located in memory location <i>operand</i> by the value stored in the accumulator and leaves the result in the accumulator.
33	DIV	Divides the integer value stored in the accumulator by the value located in memory location <i>operand</i> .
<i>Control and Transfer Operations</i>		
40	BRANCH	Unconditional jump to memory location <i>operand</i> .
41	BRANCH_NEG	If accumulator value is less than zero jump to memory location <i>operand</i> .
42	BRANCH_ZERO	If accumulator value is zero then jump to memory location <i>operand</i> .
43	HALT	Stop program execution.

Table 8-3: EISCS Machine Instructions

Sample Program: Using the instruction set given in table 8-3 you can write simple programs that will run on the EISCS Mk1 computer simulator. The following program reads two numbers from the input, multiplies them together, and writes the results to the console.

Memory Location	Instruction / Contents	Action
00	1007	Read integer into memory location 07
01	1008	Read integer into memory location 08
02	2007	Load contents of memory location 07 into accumulator
03	3208	Multiply value located in memory location 08 by value stored in accumulator. Leave result in accumulator

Memory Location	Instruction / Contents	Action
04	2109	Store value currently in accumulator to memory location 09
05	1109	Write the value located in memory location 09 to the console
06	4010	Jump to memory location 10
07		
08		
09		
10	4300	Halt program

Basic Operation: This section discusses several aspects of the EISCS computer simulation operation to assist you in completing the project.

Memory: The machine language instructions that comprise an EISCS program must be loaded into memory before the program can be executed by the simulator. Represent the computer simulator's memory as an array of integers 100 elements long.

Instruction Decoding: Instructions are fetched one at a time from memory and decoded into opcodes and operands before being executed. The following code sample demonstrates one possible decoding scheme:

```
instruction = memory[program_counter++];
operation_code = instruction / 100;
operand = instruction % 100;
```

Hints:

- Use switch/case structure to implement the instruction execution logic
- You may hard code sample programs in your simulator or allow a user to enter a program into memory via the console
- Use an array of 100 integers to represent memory

SELF-TEST QUESTIONS

1. Arrays are contiguously allocated memory elements of homogeneous data types. Explain in your own words what this means.
2. What's the difference between arrays of primitive types vs. arrays of reference types?
3. Java array types directly subclass what Java class?
4. Every Java array type has a corresponding object of this type in memory. What type is it?
5. What two interfaces do Java array types implement?
6. How do you determine the length of an array?

7. Java multidimensional arrays are _____ of _____.
8. When a multidimensional array is created which dimensions are optional and which dimensions are mandatory?
9. What is meant by the term “ragged array”?
10. What’s the purpose of the main() method String array?

REFERENCES

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*. O’Reilly and Associates, Inc. Sebastopol, CA. ISBN: 0-596-00283-1

James Gosling, et. al. *The Java™ Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, MA. ISBN: 0-201-31008-2

Jon Meyer, et. al. *Java™ Virtual Machine*. O’Reilly and Associates, Inc. Sebastopol, CA. ISBN: 1-56592-194-1

Rick Miller. *C++ For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-02-8

Donald E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Second Edition. Addison-Wesley. Reading Massachusetts. ISBN: 0-201-89685-0

NOTES

CHAPTER 9



PALMER ARRIVING

TOWARD PROBLEM ABSTRACTION: CREATING NEW DATA TYPES

LEARNING OBJECTIVES

- Describe the role abstract data types play in a Java program
- State the purpose of the Java class construct
- State the purpose of a class constructor method
- State the purpose of a static class initializer
- List the four types of Java class members to include: static fields, instance fields, static methods, and instance methods
- State the parts of a method that are included in a method's signature
- Describe how to overload a method
- State the purpose and use of method parameters
- State the difference between static class methods and instance methods
- State the difference between static class fields and instance fields
- Describe how static class fields and instance fields act as global variables within a class namespace
- State the purpose and use of the UML class diagram
- Demonstrate your ability to create classes that represent abstract data types
- Demonstrate your ability to utilize class and instance fields and methods in the design of abstract data types
- State the purpose and use of the access modifiers public, protected, and private
- Describe the scoping rules associated with fields and local method variables

INTRODUCTION

A computer program is a model of a real-world problem. But real-world problems are notoriously complex. It is impossible to capture all the details and nuances of a real-world problem in software. However, it is possible to study a real-world problem closely, identify its important points or components, and then create new software data types that represent the essential features or elements of these components. The process of selecting the essential elements of a problem with an eye towards modeling them in software is referred to as *problem abstraction*.

This chapter shows you how to approach the process of problem abstraction. Along the way you will learn more about Java classes, methods, fields, Unified Modeling Language (UML) class diagrams, and object-oriented programming. You will learn how to break functionality into logical groups to formulate methods. These methods will provide a measure of code reuse that will save you both work and time.

The material discussed here builds upon that presented in previous chapters. By now you should be very comfortable with your integrated development environment or the Java SDK. You should be able to create simple Java programs, control the flow of program execution with if, if/else, for, while, and do/while statements, and you should understand the concepts and use of single-dimensional arrays. You should also be getting pretty good at looking through the Java API for classes that can help you solve problems.

Upon completion of this chapter you will have added several powerful tools to your programmer's toolbox. These tools will enable you to write increasingly complex programs with ease.

ABSTRACTION: AMPLIFY THE ESSENTIAL – ELIMINATE THE IRRELEVANT

The process of problem abstraction is summarized nicely in the following mantra: *Amplify the Essential — Eliminate the Irrelevant*. The very nature of programming demands that a measure of simplification be performed on real-world problems. Consider for a moment the concept of numbers. Real numbers can have infinite precision. This means that in the real world numbers can have an infinite number of digits to the right of the decimal point. This is not possible in a computer with finite resources and therefore the machine representation of real numbers is only an approximation. However, for all practical purposes, an approximation is all the precision required to yield acceptable calculations. In Java, real number approximations are provided by the float and double primitive data types.

ABSTRACTION IS THE ART OF PROGRAMMING

When compared with all other aspects of programming, problem abstraction requires the most creativity. You must analyze the problem at hand, extract its essential elements, and model these in software. Also, the process of identifying what abstractions to model may entail the creation of software entities that have no corresponding counterpart in the problem domain. Have you ever heard the term “Think outside the box”? It means that to make progress you must shed your old ways of thinking. You must check your prejudices and pre-conceived notions at the door. Successful programmers have mastered the art of thinking outside, inside, over, under, to the left of, and to the right of the box. With their minds they transform real-world problems into a series of program instructions that are then executed on a machine. (*After writing this paragraph and reading it over several times I would add that successful programmers have also mastered the art of reducing real-world problems to a state that can be put inside of a box!*)

Like any form of art, the mastery of problem abstraction requires lots of practice. The only way to get lots of practice with problem abstraction is to solve lots of problems and write lots of code.

WHERE PROBLEM ABSTRACTION FITS INTO THE DEVELOPMENT CYCLE

Problem abstraction straddles the analysis and design phases of the development cycle. Project requirements may or may not be fully or adequately documented. (*In fact, on most projects, the important requirements that deeply affect the quality of the source code are not documented at all and have to be derived or deduced from existing or known requirements.*) Nonetheless, you must be able to distinguish the “signal” of the problem from its “noise”. The abstractions you choose to help model the problem in software directly influence its design (*architecture*).

CREATING YOUR OWN DATA TYPES

The end result of problem abstraction is the identification and creation of one or more new data types. These data types will interact with each other in some way to implement the solution to the problem at hand. In Java, you create a new data type by defining a new class or interface. (*Interfaces are discussed in chapter 11*) These data types can then be used by other data types, or by an application, applet, servlet, Java Server Page (JSP), Enterprise Java Bean (EJB), etc. This is referred to as design by composition. (*Design by composition is discussed in chapter 10*) The new data types created through the process of problem abstraction are referred to as *abstract data types* or *user-defined types*.

To introduce you to the process of problem abstraction and the creation of new data types I will walk you through a small case-study project. The rest of this chapter is devoted to developing the data types identified in the project specification along with a detailed discussion about the inner workings of the Java class construct.

CASE-STUDY PROJECT: WRITE A PEOPLE MANAGER PROGRAM

Figure 9-1 gives the project specification that will be used to build the program presented in this chapter.

People Manager Program

Objectives:

- Apply problem abstraction to determine essential program elements.
- Create user-defined data types using the Java class construct
- Utilize user-defined data types in a Java application
- Create and manipulate arrays of user-defined data type objects

Tasks:

- Write a program that lets you manage people. The program should let you add or delete a person when necessary. The program should also let you set and query a person's last, middle, and first names as well as their birthdate. It should also let you determine a person's age.
- Store the people objects in a single dimensional array.
- Create a separate application class that utilizes the services of a `PeopleManager` class.

Figure 9-1: People Management Program Project Specification

As you can learn from reading the project specification it offers some guidance and several hints. Let's concentrate on the tasks. First, it says that you must write a program to manage people. A full-blown people management program is obviously out of the question so our first simplification will be to put a bound on exactly what functionality is provided in the final solution. Luckily we are guided in this decision by the next sentence that says the program should focus on the following functions:

- Add a person
- Delete a person
- Set a person's first, middle, and last names
- Query a person's first, middle, and last names
- Set a person's birthdate
- Query a person's birthdate
- Query a person's age

The project specification also says that you must store people objects in a single-dimensional array. This is clear enough but where will this array reside? Again, the next sentence provides a clue. It says that you must write a sepa-

rate application that utilizes the services of a `PeopleManager` class. This is a great hint that provides you with a candidate name for one of the classes that makes up the completed program.

This will suffice for a first-pass analysis of the project specification. The trick now is to derive additional requirements that are not specifically addressed. You can begin by making some assumptions. I recommend you start by identifying the number of classes you will need to write the program. One class, `PeopleManager`, is spelled out for you in the specification. Another class is also alluded to in the last sentence and that is the application class. You could name the class anything you want but I will use the name `PeopleManagerApplication`. That should make it clear to anyone reading your code the purpose of that class.

OK, you have two classes so far: `PeopleManager` and `PeopleManagerApplication`. Since you will need people objects to work with you need to create another user-defined type named `Person`. The `Person` class will implement the functionality of a person as required to fulfill the project requirements. (*You can add additional functionality if you desire to exceed the project specification.*)

I recommend now that you make a list of the classes identified thus far and assign to them the functionality each requires. One possible list for this project is given in table 9-1.

Class Name	Functionality Required
Person	<p>The <code>Person</code> class will embody the concept of a person entity. A person will have the following attributes:</p> <ul style="list-style-type: none"> • first name • middle name • last name • gender • birthdate <p>The <code>Person</code> class will provide the capability to set and query each of its attributes as well as calculate the age of a person given their birthdate and the current date.</p>
PeopleManager	<p>The <code>PeopleManager</code> class will manage an array of <code>Person</code> objects. It will have the following attributes:</p> <ul style="list-style-type: none"> • an array of <code>Person</code> objects <p>The <code>PeopleManager</code> class will also provide the following functionality:</p> <ul style="list-style-type: none"> • add a person to the array • delete a person from the array • list the people in the array
PeopleManagerApplication	<p>The <code>PeopleManagerApplication</code> class will be the Java application class that has the <code>main()</code> method. This class will be used to test the functionality of the <code>PeopleManager</code> and <code>Person</code> classes as they are developed.</p>

Table 9-1: People Manager Program Class Responsibilities

This looks like a good start. As you progress with the design and implementation of each class, especially the `Person` and `PeopleManager` classes, you may find they require functionality not originally thought of or imagined. That's OK — software design is an iterative process. As you progress with the design and implementation of a program you gain a deeper insight or understanding of the problem you are trying to solve. This knowledge is then used to improve later versions of the software. Alright — enough soap boxing! On with the project.

The next step I recommend taking is to examine each class and see what piece of its functionality might be provided by a class from the Java API. Let's look closely at the `Person` class. The requirement to calculate a person's age means that we will have to perform some sort of date calculation. The question is: "*Is this sort of thing already done for us by the Java API?*" The answer is yes. The place to look for this sort of utility functionality is in the `java.util` package. There you will find the `Calendar` class. Take time now to familiarize yourself with the `Calendar` class as you will find it helpful in other projects as well.

This completes the analysis phase of this project. You should have a fairly clear understanding of the project requirements and the number of user-defined data types required to implement the solution. The next step I recom-

mend you take is to concentrate on the Person class and implement and test its functionality in its entirety. The Person class is the logical place to start since all the other classes depend on it.

Quick Review

Problem abstraction requires lots of programmer creativity and represents the *Art* in the *Art of Programming*. Your guiding mantra during problem abstraction is to *Amplify the Essential — Eliminate the Irrelevant*. Problem abstraction is performed in the analysis and design phase of the development cycle. The abstractions you choose to model a particular problem will directly influence a program’s design.

The end result of problem abstraction is the identification and creation of one or more new data types. The data types derived through problem abstraction are referred to as abstract data types (ADTs) or user-defined data types. User-defined data types are implemented as Java classes. These classes will interact with each other in some capacity to implement the complete problem solution.

The UML Class Diagram

Now that the three classes of the people manager project have been identified you can express their relationship to each other via a UML class diagram. The purpose of a UML class diagram is to express the static relationship between classes, interfaces, and other components of a software system. UML class diagrams are used to communicate and solidify your understanding of software designs to yourself, to other programmers, to management, and to clients. Figure 9-2 gives a basic UML diagram showing the static relationship between the classes identified in the people manager project.

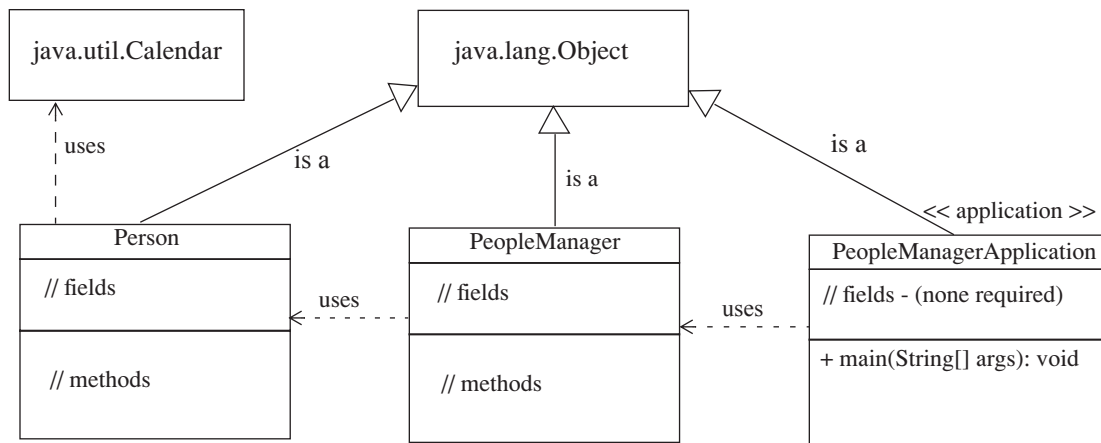


Figure 9-2: Class Diagram for People Manager Classes

Each rectangle shown in figure 9-2 represents a class. The lines tipped with the hollow arrowheads represent generalization and specialization. The arrow points from the specialized class to the generalized class. This represents an “is a...” relationship between the classes. As figure 9-2 illustrates, the classes Person, PeopleManager, and PeopleManagerApplication extend the functionality provided by the java.lang.Object. The Object class serves as the direct base class for all reference types that do not explicitly extend another class. (*Inheritance is discussed in detail in chapter 11.*)

Each class rectangle can be drawn as a simple rectangle or with three compartments. The uppermost compartment will have the class name, the middle compartment will list the fields, and the bottom compartment will list the methods.

Figure 9-2 further shows that the PeopleManagerApplication class is an application. It will have one method — main(). It could have fields and other methods since it is a Java class but in this example no other fields or methods are required.

The `PeopleManagerApplication` class uses the services of the `PeopleManager` class. This is indicated by the dashed arrow pointing from the `PeopleManagerApplication` class to the `PeopleManager` class. The dashed arrow represents a dependency. The `PeopleManager` class will have several attributes and methods which have yet to be defined.

The `PeopleManager` class will use the services of the `Person` class. The `Person` class will have attributes and methods as well. These will be developed in the next several sections.

The `Person` class uses the services of the `java.util.Calendar` class. The `Calendar` class will give the `Person` class the ability to calculate the age of each `Person` object.

Now that you have a basic design for the people manager project you can concentrate on one piece of the design and implement its functionality. Over the next several sections I will discuss the Java class construct in detail and show you how to create the `Person` and `PeopleManager` classes. Along the way I will show you how to test these classes using the `PeopleManagerApplication` class.

Quick Review

A Unified Modeling Language (UML) class diagram is used to show the static relationship between classes that participate in a software design. The class diagram is used to by programmers to express and clarify design concepts to themselves, to other programmers, to management, and to clients.

In UML a class is represented by a rectangle. The rectangle can have three compartments. The uppermost compartment contains the class name, the middle compartment contains fields, and the bottom compartment contains the methods.

Generalization and specialization is indicated with lines tipped with hollow arrows. The arrow points from the specialized class to the generalized class. The generalized class is the base class and the specialized class is the derived or subclass. Generalizations specify “is a...” relationships between base and subclasses.

Dependencies are indicated by dashed arrows pointing to the class being depended upon. Dependencies are one way to indicate “uses...” relationships between classes.

OVERVIEW OF THE JAVA CLASS CONSTRUCT

This section presents an overview of the Java class construct. Java classes consist of field and method members. This section explains the purpose of these class members and prepares you for a more detailed discussion of class concepts in later sections. You have already been exposed to the structure of a Java application class in chapter 6 so some of this material will be a review.

FOUR CATEGORIES OF CLASS MEMBERS

A Java class consists of *field* and *method* members. A field is a variable or constant that represents a class or instance attribute. Fields are used to set and maintain object state information. Fields can be *static* or *non-static*. Methods can be static (*like the main() method*) or non-static as well. The following sections describe static and non-static fields and methods in more detail.

STATIC OR CLASS-WIDE FIELDS

A static field represents an attribute that is shared among all object instances of a particular class. This means that the field values are maintained outside of any particular instance and therefore do not require a reference to an object to access. Another term used to describe static fields is *class* or *class-wide fields*. Good candidates for static fields are class constants. If you examine the Java API classes you will see wide-spread use of class constants.

NON-STATIC OR INSTANCE FIELDS

Non-static fields represent attributes for which each object has their very own copy. Another term used to describe non-static fields is *instance fields*. It's through the use of instance fields that objects can set and maintain their attribute state information. For example, if we are talking about Person objects, each Person object will have its own first name, middle name, last name, gender, and birthdate. This instance attribute state information is not shared with other Person objects. Figure 9-3 graphically illustrates the relationship between static and non-static fields.

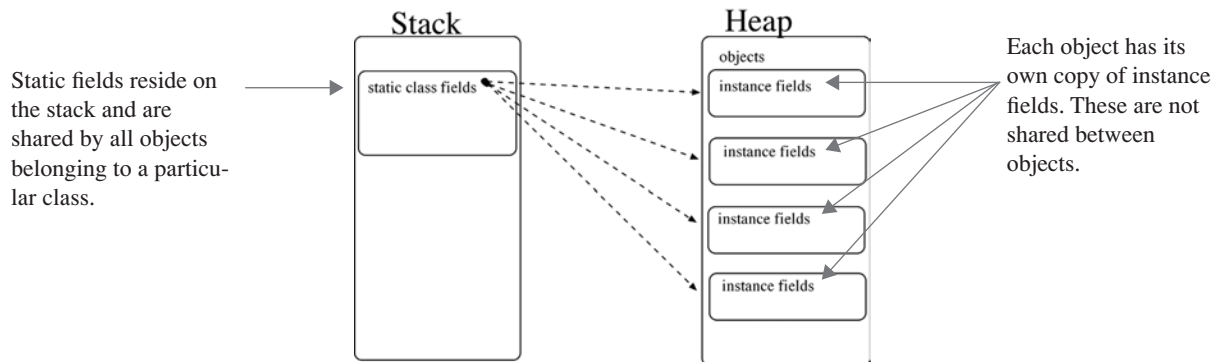


Figure 9-3: Static and Non-Static Fields

STATIC OR CLASS-WIDE METHODS

A static method can be called via the class name and has direct access only to a class's static fields. Static methods must access instance fields via an object reference. You are already familiar with the `main()` method and in chapter 6 you saw how the `main()` method accessed static and non-static fields. You will very rarely, if ever, write a static method other than `main()`. If you do write a static method it will usually be for a very specialized purpose. (See the *Singleton and Factory patterns discussed in chapter 25.*) You will see a static method in action when I show you how to use the `Calendar` class.

NON-STATIC OR INSTANCE METHODS

A non-static method enjoys direct access to a class's static and non-static fields. Another name for a non-static method is *instance method*. Instance methods must be accessed or called via an object reference, hence the name instance method. The methods you create will overwhelmingly be non-static methods.

ACCESS MODIFIERS

The access modifiers *public*, *protected*, and *private* are used to control access to fields and methods. If no access is specified then default or package access is implied.

public

The keyword `public` indicates that the class, field, or method is accessible to all clients. Generally speaking, most of the classes, and the methods you declare in a class, will be `public`. Fields, when intended to be used as class-wide constants, will often be declared `public` as well.

private

The keyword `private` indicates that the field or method is intended for internal class use only and is not available for use by client programs. You will usually declare non-static instance fields to be `private`. You can think of `private` fields as being surrounded by the protective cocoon of the class.

Methods are often declared to be private as well. Private methods are intended to be utilized exclusively by other methods within the class. These private methods are often referred to as utility methods since they are usually written to perform some utility function that is not intended to be part of the class interface.

PROTECTED

The keyword `protected` is used to prevent horizontal access to fields and methods but allow fields and methods to be inherited by subclasses. The `protected` keyword is covered in detail in chapter 11.

PACKAGE

If a class, field, or method is not declared to be public, private, or protected then it has default or package accessibility. Package accessibility is discussed in detail in chapter 11 as well.

THE CONCEPTS OF HORIZONTAL ACCESS, INTERFACE, AND ENCAPSULATION

The term *horizontal access* is used to describe the access a client object has to the methods and fields of a server object. The client object represents the code that uses the services of another object. It can do this in two ways: 1) by accessing a class's public static methods or fields, or 2) by creating an instance of the class and accessing its public non-static fields and methods via an object reference.

The fields and methods a class exposes as public are collectively referred to as its interface. Client code becomes dependent upon these interface fields and methods. The wrong kind of change to a class's interface will break any code that depends upon that interface. When changing a class's interface the rule-of-thumb is that you can add public fields and methods but they can never be removed. If you look through the Java API you will see lots of classes with deprecated methods. A deprecated method is a method that is targeted for deletion in some future version of the API. These methods are not yet removed because doing so would break existing programs that utilize (*depend upon*) those methods.

Any field or method declared private is said to be encapsulated within their class since they are shielded from horizontal access by client code. Generally speaking, a class's interface can be thought of as the set of services it provides to client programs. It provides those services by manipulating its private, or encapsulated, data structures. The idea is that at some point in the future a programmer may think up a new way to deliver a particular service's functionality. They may do this by making changes to the class's internal, or private, data structures. Since these data structures are encapsulated, a change to them will have no effect on client code, except perhaps for the effects of an improvement to the service provided.

Figure 9-4 illustrates the concept of horizontal access and the effects of using public and private access modifiers.

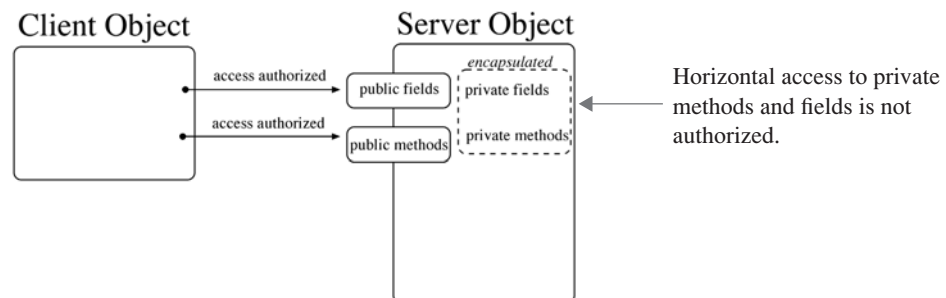


Figure 9-4: Horizontal Access Controlled via Access Modifiers public and private

The concepts of interface, horizontal access, and encapsulation are important to the world of object-oriented programming which means they are important to you. You will deal with these issues every time you write code in the Java language.

Quick Review

Java classes have four types of members: 1) static fields, 2) non-static fields, 3) static methods, and 4) non-static methods. Static fields are shared between all class objects whereas each object has its very own copy of instance fields. Alternative names for static are class and class-wide. An alternative name for non-static is instance. Static methods can only directly manipulate static fields. Non-static methods can manipulate both static and non-static fields.

There are three access modifiers: public, private, and protected. All three access modifiers are used to control horizontal access. If no access is specified the default or package access is implied.

Horizontal access is the term used to describe how a client object uses the services of a server object. Public fields and methods published by a class are collectively referred to as its interface. Private fields and methods are said to be encapsulated within the class. Client code becomes dependent upon a class's interface. Changes to a class's interface can potentially break any client code that depends upon its interface.

Methods

A method is a named module of executable program functionality. A method can contain program statements that, when grouped together, represent a basic level of code reuse. You access the functionality of a method by calling the method using its name in a program. (*I use the term program here to mean any piece of code that could possibly use the services of the class that defines the method. This might include 1) another method within the class you are defining, 2) another class within your program, or 3) a third-party program that wants to use the services provided by your program.*)

In the Java language a method must belong to a class; they cannot exist or be defined outside of a class construct.

METHOD NAMING – USE ACTION WORDS THAT INDICATE THE METHOD'S PURPOSE

When naming a method use action words (verbs) that provide an indication of the method's intended purpose. I discussed method naming guidelines in chapter 1 in the section titled *Identifier Naming — Writing Self-Commenting Code*.

MAXIMIZE METHOD COHESION

The first rule-of-thumb to keep in mind when writing a method is to keep the functionality of the method focused on the task at hand. The formal term used to describe a method's focus characteristic is *cohesion*. The goal is to write highly-cohesive methods. A method that does things it really shouldn't be doing is not focused and is referred to as minimally cohesive. You can easily write cohesive methods if you follow this two-step approach:

- Step 1 - Follow the advice offered in the previous sub-section and start with a good method name.
The name of the method must indicate the method's intended purpose.
- Step 2 - Keep the method's body code focused on performing the task indicated by the method's name. A well-named, maximally-cohesive method will pull no surprises!

Sounds simple enough — but if you're not careful you can slip functionality into a method that doesn't belong there. Sometimes you will do this because you are lazy, and sometimes it will happen no matter how hard you try to avoid doing so. Practice makes perfect!

STRUCTURE OF A METHOD DEFINITION

A method definition declares and implements a method. A method definition consists of several optional and mandatory components. These include method modifiers, a return type or void, method name, parameter list, and throws clause. I discuss these method components in detail below. Figure 9-5 shows the structure of a method definition.


```

method_modifiersopt return_type or voidopt method_name( parameter_listopt ) throwsopt {
    // method body - program statements go here
}

```

Figure 9-5: Method Definition Structure

Any piece of the method definition structure shown in figure 9-5 that's labeled with the subscript *opt* is optional and can be omitted from a method definition depending on the method's required behavior. In this chapter I will focus on just a few of the potentially many method variations you can write. You will be gradually introduced to different method variations and as you progress through the book. The following sections describe each piece of the method definition structure in more detail.

Method Modifiers (optional)

Use method modifiers to specify a particular aspect of method behavior. Table 9-2 lists and describes the Java keywords that can be used as method modifiers.

Modifier	Description
public	The public keyword declares the method's accessibility to be public. Public methods can be accessed by client code. (<i>i.e.</i> , grants horizontal access to the method)
protected	The protected keyword declares the method's accessibility to be protected. Protected accessibility prevents horizontal access but allows the method to be inherited by derived classes.
private	The private keyword declares the method's accessibility to be private. It prevents both horizontal access and method inheritance.
abstract	The abstract keyword is used to declare a method that contains no body (<i>no implementation</i>). The purpose of an abstract method is to defer the implementation of a method's functionality to a subclass. Abstract methods are discussed in chapter 11.
static	The static keyword is used to create a static or class method.
final	The final keyword is used to specify that a method cannot be overridden or hidden by a subclass method. Final methods are covered in chapter 11.
synchronized	The synchronized keyword is used to specify that a method requires a lock before it executes. Synchronized methods are used in multi-threaded programming and are covered in detail in chapter 16.
native	The native keyword is used to declare a native method. A native method serves as a declaration only and the implementation of the method will be written in another programming language like C or C++. Native method code executes directly on the hardware vice in the Java virtual machine environment.
strictfp	The strictfp keyword makes all float or double expressions within the body of the method explicitly FP-strict. For more information about the strictfp keyword or FP-strict expressions refer to The Java Language Specification, Second Edition which is listed in the references section at the end of this chapter.

Table 9-2: Java Method Modifier Keywords

RETURN TYPE OR void (OPTIONAL)

A method can return a result as a side effect of its execution. If you intend for a method to return a result you must specify the return type of the result. If the method does not return a result then you must use the keyword “void”.

The return type and void are optional because constructor methods return neither. Constructor methods are discussed in detail below.

METHOD NAME (MANDATORY)

The method name is mandatory. As was discussed earlier you should use verbs in method names since methods perform some sort of action. If you chose to ignore good method naming techniques you will find that your code is hard, if not impossible, to read and understand and as a result will also be hard to fix if it’s broken.

PARAMETER LIST (OPTIONAL)

A method can specify one or more formal parameters. Each formal parameter has a type and a name. The name of the parameter has local scope within the body of the method and hides any field members having the same name.

METHOD BODY (OPTIONAL FOR ABSTRACT OR NATIVE METHODS)

The method body is denoted by a set of opening and closing brackets. Any code that appears between a method’s opening and closing brackets is said to be in the body of the method. If you are declaring an abstract or native method you will omit the braces and terminate the method declaration with a semicolon.

EXAMPLE METHOD DEFINITIONS

This section offers a few examples of method definitions. The body code is omitted so that you can focus on the structure of each method definition. The following line of code would define a method that returns a String object that represents the first name of some object (*perhaps a Person object*).

```
public String getFirstName() { // body code goes here }
```

Notice that the above method definition uses the public access modifier, declares a return type of String, and takes no arguments because it declares no parameters. The name of the method is getFirstName which does a good job of describing the method’s purpose.

The next method declaration might be used to set an object’s first name:

```
public void setFirstName(String first_name) { // body code goes here }
```

This method is also public but it does not return a result hence the use of the keyword void. It contains one parameter named first_name that is of type String.

The following method definition might be used to get a Person object’s age:

```
public int getAge() { // body code goes here }
```

This method is public and returns an integer primitive type result. It takes no arguments. See if you can guess what type of method is being defined by the following definition:

```
public Person(String f_name, String m_name, String l_name) {
    // body code goes here
}
```

If you guessed that it was a constructor method you would be right. Constructor methods have no return type, not even void. This particular constructor declares three formal parameters having type String. Constructors are discussed in detail below.

METHOD SIGNATURES

Methods have a distinguishing characteristic known as a signature. A method's signature consists of its name and the number and types of its parameters. Method modifiers and return types are not part of a method's signature. It's important to understand the concept of method signature so that you can understand the concept of method overloading which is discussed in the next section.

Methods with different names and the same parameter list have different signatures. Methods with the same name and different parameter lists have different signatures as well and are said to be overloaded (because they share the same name). Methods cannot have the same name and identical parameter lists. This will cause a compiler error.

OVERLOADING METHODS

A class can define more than one method with the same name but having different signatures. This is referred to as method overloading. You would overload methods when the method performs the same function but in a slightly different way or on different argument types. The most commonly overloaded method is the class constructor. You will see many examples of overloaded class constructors throughout the remaining chapters of this book.

Another frequently encountered method overloading scenario occurs when you want to provide a public method for horizontal access but actually do the work behind the scenes with a private method. The only rule, as stated above, is that each method must have a different signature, which means their names can be the same but their parameter lists must be different in some way. The fact that one is public and the other is private has no bearing on their signatures.

CONSTRUCTOR METHODS

Constructor methods are special methods whose purpose is to set up or build the object in memory when it is created. You can choose not to define a constructor method for a particular class if you desire. In this case the Java compiler will create a default constructor for you. This default constructor will usually not provide the level of functionality you require except perhaps in the case of very simple or trivial class declarations. If you want to be sure of the state of an object when it is created you must define one or more constructor methods.

Quick Review

Methods are named modules of executable program functionality. Methods contain program statements that, when grouped together, represent a basic level of code reuse. You access the functionality of a method by calling the method using its name in a program.

Methods should be well-named and maximally cohesive. A well-named, maximally-cohesive method will pull no surprises!

Method definitions have structure. Their behavior can be optionally modified with method modifiers, they can optionally specify a return result type or void, and they can have an optional parameter list.

Methods have a distinguishing characteristic known as a method signature. Methods with different names and parameter lists are said to have different signatures. Methods with different names and the same parameter list also have different signatures. Methods with the same name and different parameter lists have different signatures as well and are said to be overloaded (*because they share the same name*). Methods cannot have the same name and identical parameter lists. This will cause a compiler error.

Constructor methods are used to set up or build an object when it's created in memory. Java will provide a default constructor but it may or may not provide the level of functionality you require.

BUILDING AND TESTING THE PERSON CLASS

Now that you have been introduced to the Java class construct in more detail it's time to apply some of what was discussed above to create and test the Person class. To get the Person class in working order you will have to take the results of the analysis performed earlier and map attributes and functionality to fields and methods. As you build the Person class you may discover that you need to add more fields or methods to fully implement the class to your satisfaction. That's a normal part of the programming process.

To write the code for the Person class I will use the development cycle presented in chapter 1 and explained in detail in chapter 3. The development cycle is applied iteratively. (*i.e.*, *I will apply the steps of plan, code, test, and integrate repeatedly until I have completed the code.*)

START BY CREATING THE SOURCE FILE AND CLASS DEFINITION SHELL

I recommend you start this process by creating the Person.java source file and the Person class definition shell. Example 9.1 gives the code for this early stage of the program.

9.1 Person.java
(1st Iteration)

```
3     public class Person {
4
5     } //end Person class
```

At this point I recommend you compile the code to ensure you've typed everything correctly and that the name of the class matches the name of the file. The next thing to do would be to refer to table 9-1 and see what attributes or fields the Person class must contain and add those fields. This is done in the next section.

DEFINING PERSON INSTANCE FIELDS

Referring to table 9-1 you learn that the Person class represents a person entity in our problem domain. Each person has their own name, gender, and birthdate so these are good candidates for instance fields in the Person class.

Example 9.2 shows the Person class code after the instance fields have been added.

9.2 Person.java
(2nd Iteration)

```
1     import java.util.*;
2
3     public class Person {
4         private String first_name = null;
5         private String middle_name = null;
6         private String last_name = null;
7         private Calendar birthday = null;
8         private String gender= null;
9
10    } //end Person class
```

Two Java API classes are used for the fields in example 9.2: String and Calendar. The import statement is used on line 1 to provide shortcut naming for the Calendar class. These fields represent a first attempt at defining fields for the Person class. Each field is declared to be private which means they will be encapsulated by the Person class to prevent horizontal access. The only way to access or modify these fields will be through the Person class's public interface methods. Let's define a few of those right now. But, before you move on, compile the Person.java source file again to make sure you didn't break anything.

DEFINING PERSON INSTANCE METHODS

Now that several Person class instance fields have been defined it's time to define a few methods. After you've defined several methods you can use them to test those aspects of Person class behavior. A good place to start is with the constructor method and several getter and setter methods. (*i.e.*, *methods whose names begin with get and set.*)

The Constructor Method

A good method to define first is the constructor method. Remember, the purpose of the constructor method is to properly initialize an object when it is created in memory. In the case of the Person class this means that each person object's fields must be initialized to some valid value. To make this happen I will add a constructor method that takes a parameter list that matches the fields contained in the Person class. These parameters will then be used to initialize each field. Example 9.3 gives the code for the Person class definition after the constructor has been added.

9.3 Person.java
(3rd Iteration)

```

1      import java.util.*;
2
3      public class Person {
4          private String first_name = null;
5          private String middle_name = null;
6          private String last_name = null;
7          private String date_of_birth = null;
8          private Calendar birthday = null;
9          private String gender = null;
10
11         public Person(String f_name, String m_name, String l_name,
12                        int dob_year, int dob_month, int dob_day, String gender){
13             first_name = f_name;
14             middle_name = m_name;
15             last_name = l_name;
16             this.gender = gender;
17
18             birthday = Calendar.getInstance();
19             birthday.set(dob_year, dob_month, dob_day);
20         }
21     } // end Person class
22

```

The Person constructor method begins on line 11. Notice that it is declared to be public and has no return value. It has seven parameters. The first three parameters, `f_name`, `m_name`, and `l_name` are used on lines 13, 14, and 15 to initialize the instance fields `first_name`, `middle_name`, and `last_name`.

The next three constructor parameters: `dob_year`, `dob_month`, and `dob_day` are used to initialize a Person object's birthday Calendar object. This is done on line 19 after the `Calendar.getInstance()` method call is used to initialize the birthday reference. (*If you haven't already studied the Calendar class functionality you might want to do so now.*)

The last parameter, `gender`, is used on line 16 to initialize the instance field having the same name. Since the parameter name `gender` will hide the field named `gender` it is required to prefix the field name with the `this` keyword so the compiler can tell the difference between the field and the parameter. There are other ways you could handle this situation. For example, you could start each field name with an underscore `'_'` character so that when you refer to a field in a method you and the compiler would know the difference between fields and local method variables. If you used this approach the field named `gender` would become `_gender` and the assignment on line 16 would then take the following form:

```
_gender = gender;
```

OK, now that you've got the constructor written compile the Person.java source file to ensure you didn't break anything. It's now time to add a few more methods so we can test this puppy.

Adding A Few Accessor Methods

Accessor methods are methods that simply return some sort of object state information. Accessor methods do not alter or modify the fields they access or do anything else that might change the state of the object. A good place to start would be to add a few accessor methods to return a Person object's first name, middle name, last name, and gender. These methods will be easy to write. Once these methods are in place we can use the Person class in another program and test its functionality. Example 9.4 shows the Person class after these accessor methods have been added.

9.4 Person.java
(4th Iteration)

```

1      import java.util.*;
2
3      public class Person {
4          private String first_name = null;
5          private String middle_name = null;
6          private String last_name = null;
7          private Calendar birthday = null;
8          private String gender = null;
9
10         public Person(String f_name, String m_name, String l_name,
11             int dob_year, int dob_month, int dob_day, String gender){
12             first_name = f_name;
13             middle_name = m_name;
14             last_name = l_name;
15             this.gender = gender;
16             birthday = Calendar.getInstance();
17             birthday.set(dob_year, dob_month, dob_day);
18         }
19
20         public String getFirstName(){ return first_name;}
21
22         public String getMiddleName(){ return middle_name;}
23
24         public String getLastName(){ return last_name; }
25
26         public String getGender(){ return gender;}
27
28     } //end Person class

```

The accessor methods appear on lines 20, 22, 24, and 26. Each method returns a String object and each method name begins with the word *get* followed by the name of the particular property being retrieved. The convention of preceding accessor methods (*also referred to as getters*) with the word *get* followed by the property name beginning with a capital letter is not arbitrary. This convention conforms to the Java Beans specification. For now, I recommend you adopt the convention. Doing so will make the transition to professional Java programmer much easier.

With the constructor and getter methods in place compile the source code and get ready for testing.

TESTING THE PERSON CLASS

Testing the Person class at this stage of the development cycle will consist of creating a Person object and calling each getter method. This will effectively test the constructor and accessor methods.

Use The PeopleManagerApplication Class As A Test Driver

To test the Person class functionality you'll need to create another class that's a Java application. Since you need to create the PeopleManagerApplication class anyway you might as well use that class as a test driver. (*The term driver means a small program written specifically to run or test another program.*) Example 9.5 gives the code for the PeopleManagerApplication class with a few lines of code that tests the functionality of the Person class developed thus far. Figure 9-6 shows the results of running this program.

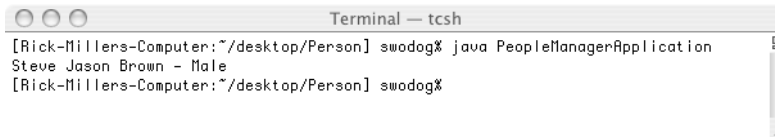
9.5 PeopleManagerApplication.java
(Testing Person)

```

1      public class PeopleManagerApplication {
2          public static void main(String[] args){
3              Person p1 = new Person("Steve", "Jason", "Brown", 1980, 3, 4, "Male");
4
5              System.out.println(p1.getFirstName() + " " + p1.getMiddleName()
6                  + " " + p1.getLastName() + " - " + p1.getGender());
7          } // end main
8      } // end PeopleManagerApplication class

```

Everything appears to run fine. It's now time to add a few more features to the Person class.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/Person] swodog% java PeopleManagerApplication
Steve Jason Brown - Male
[Rick-Millers-Computer:~/desktop/Person] swodog%
```

Figure 9-6: Results of Running Example 9.5

Adding FEATURES TO THE PERSON CLASS – CALCULATING AGE

Returning to table 9-1 for some direction reveals the requirement to calculate a person's age. This could be done in several ways. Reflect for a moment on how you get the age of a real live person. You might ask them for their birth date and perform the calculation yourself. You would have to know today's date and subtract their birth date to yield the answer. Another way would be to just ask the person how old they are. They would do the calculation for you and they would have to know today's date to do that. For this example I am going to go with the last scenario. I will add an accessor method named `getAge()` that computes the person's age and returns the result. Example 9.6 shows the modified Person class code.

*9.6 Person.java
(5th Iteration)*

```
1      import java.util.*;
2
3      public class Person {
4          private String first_name = null;
5          private String middle_name = null;
6          private String last_name = null;
7          private Calendar birthday = null;
8          private String gender = null;
9
10         public static final String MALE = "Male";
11         public static final String FEMALE = "Female";
12
13         public Person(String f_name, String m_name, String l_name, int dob_year,
14             int dob_month, int dob_day, String gender){
15             first_name = f_name;
16             middle_name = m_name;
17             last_name = l_name;
18             this.gender = gender;
19
20             birthday = Calendar.getInstance();
21             birthday.set(dob_year, dob_month, dob_day);
22         }
23
24         /***** Added the getAge() method *****/
25         public int getAge(){
26             Calendar today = Calendar.getInstance();
27             int now = today.get(Calendar.YEAR);
28             int then = birthday.get(Calendar.YEAR);
29             return (now - then);
30         }
31
32         public String getFirstName(){ return first_name; }
33
34         public String getMiddleName(){ return middle_name; }
35
36         public String getLastName(){ return last_name; }
37
38         public String getGender(){ return gender; }
39
40     } //end Person class
```

After making the necessary modifications to the Person class you can test the changes in the `PeopleManagerApplication` class. Example 9.7 shows the code for the modified `PeopleManagerApplication` class. Figure 9-7 shows the results of running this program.

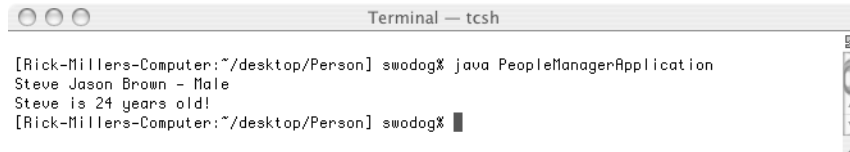
*9.7 PeopleManagerApplication.java
(Testing Person age functionality)*

```
1      public class PeopleManagerApplication {
2          public static void main(String[] args){
3              Person p1 = new Person("Steve", "Jason", "Brown", 1980, 3, 4, "Male");
4          }
```

```

5      System.out.println(p1.getFirstName() + " " + p1.getMiddleName()
6          + " " + p1.getLastName() + " - " + p1.getGender());
7      System.out.println(p1.getFirstName() + " is " + p1.getAge() + " years old!");
8
9      } // end main
10     } // end PeopleManagerApplication class

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/Person] swodog% java PeopleManagerApplication
Steve Jason Brown - Male
Steve is 24 years old!
[Rick-Millers-Computer:~/desktop/Person] swodog% █

```

Figure 9-7: Results of Running Example 9.7

Adding Features To The Person Class – Convenience Methods

The age feature seems to work pretty good. However, from looking at the code in example 9.7 it sure looks like a hassle to get a Person object's full name. It might be a good idea to add an accessor method that will do the job for you. While you're at it you could add a method that returns both the full name and age. Example 9.8 shows the modified Person class.

*9.8 Person.java
(6th Iteration)*

```

1      import java.util.*;
2
3      public class Person {
4          private String first_name = null;
5          private String middle_name = null;
6          private String last_name = null;
7          private Calendar birthday = null;
8          private String gender = null;
9
10         public static final String MALE = "Male";
11         public static final String FEMALE = "Female";
12
13         public Person(String f_name, String m_name, String l_name, int dob_year,
14             int dob_month, int dob_day, String gender){
15             first_name = f_name;
16             middle_name = m_name;
17             last_name = l_name;
18             this.gender = gender;
19
20             birthday = Calendar.getInstance();
21             birthday.set(dob_year, dob_month, dob_day);
22         }
23
24         public int getAge(){
25             Calendar today = Calendar.getInstance();
26             int now = today.get(Calendar.YEAR);
27             int then = birthday.get(Calendar.YEAR);
28             return (now - then);
29         }
30
31         public String getFullName(){ return (first_name + " " + middle_name + " " + last_name); }
32
33         public String getFirstName(){ return first_name; }
34
35         public String getMiddleName(){ return middle_name; }
36
37         public String getLastName(){ return last_name; }
38
39         public String getNameAndAge(){ return (getFullName() + " " + getAge()); }
40
41         public String getGender(){ return gender; }
42
43     } //end Person class

```

Referring to example 9.8 — the `getFullName()` method appears on line 31. It simply concatenates the fields `first_name`, `middle_name`, and `last_name` with spaces and returns the resulting String object that represents the Person object's full name.

The `getNameAndAge()` method on line 39 utilizes the services of the `getFullName()` and `getAge()` methods. This is a good example of code reuse at the class level. Since the methods exist and already provide the required functionality it's a good idea to use them.

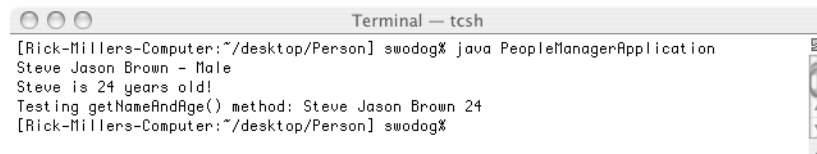
It's time to compile the Person class and test the changes. Example 9.9 gives the modified `PeopleManagerApplication` class with the changes required to test the Person class's new functionality. Notice the code is a lot cleaner now. Figure 9-8 shows the results of running this program.

9.9 *PeopleManagerApplication.java*
(Testing new Person functionality)

```

1      public class PeopleManagerApplication {
2          public static void main(String[] args){
3              Person p1 = new Person("Steve", "Jason", "Brown", 1980, 3, 4, "Male");
4
5              System.out.println(p1.getFullName() + " - " + p1.getGender());
6              System.out.println(p1.getFirstName() + " is " + p1.getAge() + " years old!");
7              System.out.println("Testing getNameAndAge() method: " + p1.getNameAndAge());
8
9          } // end main
10     } // end PeopleManagerApplication class

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/Person] swodog% java PeopleManagerApplication
Steve Jason Brown - Male
Steve is 24 years old!
Testing getNameAndAge() method: Steve Jason Brown 24
[Rick-Millers-Computer:~/desktop/Person] swodog%

```

Figure 9-8: Results of Running Example 9.9

Adding FEATURES TO THE PERSON CLASS – FINISHING TOUCHES

It looks like the Person class is beginning to take shape. Referring to table 9-1 for some final advice reveals that you must be able to set each of the Person attributes as well as query them. This is accomplished by adding mutator or setter methods. Setter methods begin with the word *set* followed by the property name they are setting. If you are building the capability into a class to get and set a property then the property name used in the getter method will match the property name in the corresponding setter method.

In addition to the setter methods I will add a set of Person class-wide constants to represent the genders Male and Female. These will be String constants that can be used in the Person constructor call to supply the correctly formed string to the gender field.

The getter method `getAge()` will not have a corresponding setter method. (*i.e., I will not write a `setAge()` method.*) I will instead write a `setBirthday()` method that will set the year, month, and day of a Person object's birthday Calendar object. Then, when the `getAge()` method is next called it will calculate the new age.

Example 9.10 gives the modified Person class source code.

9.10 *Person.java*
(Final Iteration)

```

1      import java.util.*;
2
3      public class Person {
4          private String first_name = null;
5          private String middle_name = null;
6          private String last_name = null;
7          private Calendar birthday = null;
8          private String gender = null;
9
10         public static final String MALE = "Male";
11         public static final String FEMALE = "Female";
12
13         public Person(String f_name, String m_name, String l_name, int dob_year,
14                       int dob_month, int dob_day, String gender){
15             first_name = f_name;
16             middle_name = m_name;
17             last_name = l_name;
18             this.gender = gender;
19
20             birthday = Calendar.getInstance();

```

```

21     birthday.set(dob_year, dob_month, dob_day);
22 }
23
24 public int getAge(){
25     Calendar today = Calendar.getInstance();
26     int now = today.get(Calendar.YEAR);
27     int then = birthday.get(Calendar.YEAR);
28     return (now - then);
29 }
30
31 public String getFullName(){ return (first_name + " " + middle_name + " " + last_name); }
32
33 public String getFirstName(){ return first_name; }
34 public void setFirstName(String f_name) { first_name = f_name; }
35
36 public String getMiddleName(){ return middle_name; }
37 public void setMiddleName(String m_name){ middle_name = m_name; }
38
39 public String getLastName(){ return last_name; }
40 public void setLastName(String l_name){ last_name = l_name; }
41
42 public String getNameAndAge(){ return (getFullName() + " " + getAge()); }
43
44 public String getGender(){ return gender; }
45 public void setGender(String gender){ this.gender = gender; }
46
47 public void setBirthday(int year, int month, int day){ birthday.set(year, month, day); }
48
49 } //end Person class

```

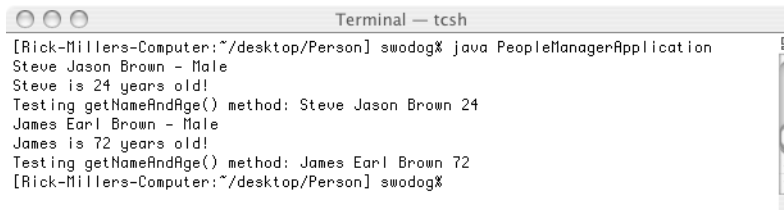
Example 9.11 shows the new Person class functionality being tested in the PeopleManagerApplication class. Figure 9-9 shows the results of running this program.

*9.11 PeopleManagerApplication.java
(Final Person test)*

```

1 public class PeopleManagerApplication {
2     public static void main(String[] args){
3         Person p1 = new Person("Steve", "Jason", "Brown", 1980, 3, 4, Person.MALE);
4
5         System.out.println(p1.getFullName() + " - " + p1.getGender());
6         System.out.println(p1.getFirstName() + " is " + p1.getAge() + " years old!");
7         System.out.println("Testing getNameAndAge() method: " + p1.getNameAndAge());
8
9         p1.setFirstName("James");
10        p1.setMiddleName("Earl");
11        p1.setLastName("Brown");
12        p1.setBirthday(1932, 8, 9);
13
14        System.out.println(p1.getFullName() + " - " + p1.getGender());
15        System.out.println(p1.getFirstName() + " is " + p1.getAge() + " years old!");
16        System.out.println("Testing getNameAndAge() method: " + p1.getNameAndAge());
17    } // end main
18 } // end PeopleManagerApplication class

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/Person] swodog% java PeopleManagerApplication
Steve Jason Brown - Male
Steve is 24 years old!
Testing getNameAndAge() method: Steve Jason Brown 24
James Earl Brown - Male
James is 72 years old!
Testing getNameAndAge() method: James Earl Brown 72
[Rick-Millers-Computer:~/desktop/Person] swodog%

```

Figure 9-9: Results of Running Example 9.11

Quick Review

Abstract data types can be incrementally built and tested by iteratively applying the steps of the development cycle. Start with the class definition shell and then add fields and methods as required to fulfill the class's design objectives.

Class functionality can be testing with the help of a test driver. A test driver is a small program that's used to exercise the functionality of another program.

BUILDING AND TESTING THE PEOPLEMANAGER CLASS

Now that the Person class is finished it's time to shift focus to the PeopleManager class. Referring to table 9-1 again reveals that the PeopleManager class will manage an array of Person objects and be able to insert Person objects into the array, delete Person objects from the array, and list the names and perhaps other information for Person objects contained in the array at any particular instant.

The same approach used to develop the Person class will be used here to develop the PeopleManager class. The development cycle will be applied iteratively to yield the final result.

DEFINING THE PEOPLEMANAGER CLASS SHELL

Example 9.12 gives the source code for the PeopleManager class definition shell. Compile the code to ensure the class name matches the source file name and to catch any early structural mistakes.

9.12 PeopleManager.java
(1st Iteration)

```
1     public class PeopleManager {
2
3
4
5     } // end PeopleManager class
```

To this shell you will add fields and methods.

DEFINING PEOPLEMANAGER FIELDS

Table 9-1 says the PeopleManager class will manage an array of Person objects. This means it will need a field that is a single-dimensional array of type Person. Do not actually create the array. Instead, initialize the array reference to null like the fields of the Person class. Example 9.13 gives the modified source code for the PeopleManager class after the declaration of the people array.

9.13 PeopleManager.java
(2nd Iteration)

```
1     public class PeopleManager {
2         Person[] people_array = null;
3
4
5     } // end PeopleManager class
```

Additional fields may be required but for now this is a good start. Time to add some methods.

DEFINING PEOPLEMANAGER CONSTRUCTOR METHODS

You can start with the constructor method. In fact I'll show you how to overload the constructor method to create a default constructor. (*Remember, a default constructor is one that has no parameters.*) The PeopleManager constructor will need to create the people_array object. To do this it will need to know how long the array must be. You will supply the length via a constructor parameter. If you do not supply a length argument when you create an instance of PeopleManager then the default constructor will create the people_array with some default length value. In this example I will use a default length of 10. Example 9.14 gives the modified PeopleManager class after the constructors have been added.

9.14 PeopleManager.java
(3rd Iteration)

```
1     public class PeopleManager {
2         Person[] people_array = null;
3
4         public PeopleManager(int size){
```

```

5         people_array = new Person[size];
6     }
7
8     public PeopleManager(){
9         this(10);
10    }
11    }// end PeopleManager class

```

The constructor on line 4 takes an integer parameter named `size` and uses it to dynamically create the `people_array` in memory. The default constructor starts on line 8. It takes no parameters. On line 9 it calls the constructor defined on line 4 via the peculiar-looking `this(10)` call. The compiler will sort out which constructor `this()` refers to by examining the parameter list. Since there is a constructor defined to take an integer as a parameter it will use that constructor.

Compile the code to ensure you didn't break anything and then add another method so you can start seriously testing the `PeopleManager` class.

DEFINING ADDITIONAL PEOPLEMANAGER METHODS

I recommend adding the capability to add `Person` objects to the `people_array` first. Then you can add the capability to list their information, and finally, the capability to delete them.

A good candidate name for a method that adds a person would be `addPerson()`. Likewise, a good candidate name for a method that lists the `Person` objects in the array might be `listPeople()`. Example 9.15 gives the source code for the `PeopleManager` class containing the newly created `addPerson()` and `listPeople()` methods.

9.15 *PeopleManager.java*
(4th Iteration)

```

1     public class PeopleManager {
2         Person[] people_array = null;
3         int index = 0;
4
5         public PeopleManager(int size){
6             people_array = new Person[size];
7         }
8
9         public PeopleManager(){
10            this(10);
11        }
12
13        public void addPerson(String f_name, String m_name, String l_name,
14                               int dob_year, int dob_month, int dob_day, String gender){
15            if(people_array[index] == null){
16                people_array[index++] = new Person(f_name, m_name, l_name,
17                                                    dob_year, dob_month, dob_day, gender);
18            }
19            if(index >= people_array.length){
20                index = 0;
21            }
22        }
23
24        public void listPeople(){
25            for(int i = 0; i < people_array.length; i++){
26                if(people_array[i] != null){
27                    System.out.println(people_array[i].getNameAndAge());
28                }
29            }
30        }
31    }// end PeopleManager class

```

Let's look at the `addPerson()` method for a moment. It has a parameter list that contains all the elements required to create a `Person` object. These include `f_name`, `m_name`, `l_name`, `dob_year`, `dob_month`, `dob_day`, and `gender`. The first thing the `addPerson()` method does is to check to see if a particular array element is equal to null. (*The first time the `addPerson()` method is called on a particular `PeopleManager` object all the `people_array` elements will be null.*) The variable `index` has been added as an instance field so that its value is maintained.

On line 19 the value of `index` is checked to see if it is greater than or equal to the length of the array. If it is its value is set to 0.

In this simple example the `addPerson()` method will add `Person` objects to the array until the array is full. From then on it will only insert `Person` objects if the array element it's trying to access is null. (*There are better ways to implement this method and they are left as exercises at the end of the chapter.*)

The `listPerson()` method simply iterates over the `people_array` and, if the array element is not null, meaning it points to a `Person` object, it will call the `getNameAndAge()` method on the `Person` object and print the resulting `String` to the console.

TESTING THE PEOPLEMANAGER CLASS

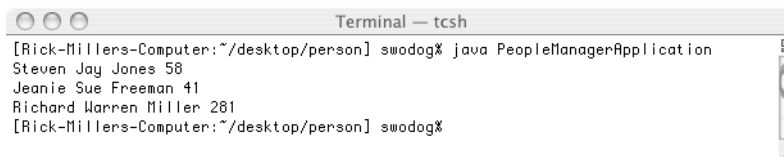
You can use the `PeopleManagerApplication` class once again to test the functionality of the `PeopleManager` class. Example 9.16 gives the source code for the modified `PeopleManagerApplication` class. Figure 9-10 shows the results of running this program.

9.16 *PeopleManagerApplication.java*
(Testing the *PeopleManager* class)

```

1      public class PeopleManagerApplication {
2          public static void main(String[] args){
3              PeopleManager pm = new PeopleManager(); // default constructor test
4
5              pm.addPerson("Steven", "Jay", "Jones", 1946, 8, 30, Person.MALE);
6              pm.addPerson("Jeanie", "Sue", "Freeman", 1963, 10, 10, Person.FEMALE);
7              pm.addPerson("Richard", "Warren", "Miller", 1723, 2, 29, Person.MALE);
8
9              pm.listPeople();
10
11         } // end main
12     } // end PeopleManagerApplication class

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/person] swodog% java PeopleManagerApplication
Steven Jay Jones 58
Jeanie Sue Freeman 41
Richard Warren Miller 281
[Rick-Millers-Computer:~/desktop/person] swodog%

```

Figure 9-10: Results of Running Example 9.16

Referring to example 9.16 — the `PeopleManager` default constructor is tested on line 3. This also tests the other `PeopleManager` constructor. (*Killed two birds with one stone here!*) The `addPerson()` method is tested on lines 5 through 7 and the `listPeople()` method is tested on line 9. Everything appears to work as expected. You can now add the capability to delete `Person` objects and perhaps some other functionality.

ADDING FEATURES TO THE PEOPLEMANAGER CLASS

The `PeopleManager` class now implements two out of three required features. You can add people to the `people_array` and you can list information about each person contained in the `people_array`. It's now time to implement the capability to delete `Person` objects from the array. A good candidate name for a method to delete a `Person` object from the array is `deletePerson()`. (*See — method naming isn't so hard!*) But wait — not so fast with the kudos. You just can't delete a `Person` from an arbitrary element. It might be better instead to delete a `Person` object from a specific `people_array` element, in which case you might want to better name the method `deletePersonAtIndex()`.

While you're at it you might want to add the capability to insert `Person` objects into any element within the array. A good candidate name for such a method might be `insertPersonAtIndex()`. Example 9.17 gives the source code for the modified `PeopleManager` class.

9.17 *PeopleManagerClass.java*
(5th Iteration)

```

1      public class PeopleManager {
2          Person[] people_array = null;
3          int index = 0;
4
5          public PeopleManager(int size){
6              people_array = new Person[size];
7          }
8

```

```

9      public PeopleManager(){
10         this(10);
11     }
12
13     public void addPerson(String f_name, String m_name, String l_name,
14                          int dob_year, int dob_month, int dob_day, String gender){
15         if(people_array[index] == null){
16             people_array[index++] = new Person(f_name, m_name, l_name,
17                                                dob_year, dob_month, dob_day, gender);
18         }
19         if(index >= people_array.length){
20             index = 0;
21         }
22     }
23
24     public void deletePersonAtIndex(int index){
25         assert((index >= 0) && (index < people_array.length));
26         people_array[index] = null;
27     }
28
29     public void insertPersonAtIndex(int index, String f_name, String m_name, String l_name,
30                                    int dob_year, int dob_month, int dob_day, String gender){
31         assert((index >= 0) && (index < people_array.length));
32         people_array[index] = new Person(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender);
33     }
34
35     public void listPeople(){
36         for(int i = 0; i < people_array.length; i++){
37             if(people_array[i] != null){
38                 System.out.println(people_array[i].getNameAndAge());
39             }
40         }
41     }
42 } // end PeopleManager class

```

Examine closely for a moment the `deletePersonAtIndex()` method. It declares one parameter named `index`. This parameter name will hide the field named `index` which is the desired behavior in this case. There is also a danger that the argument used in the `deletePersonAtIndex()` method call might be invalid given the length of the `people_array`. The `assert` mechanism is used on line 25 to enforce the precondition that the value of the `index` parameter must be greater than or equal to zero or less than the length of the `people_array`. The `assert` mechanism is also used in the `insertPersonAtIndex()` method on line 31.

The `assert` mechanism is normally off or unrecognized when you compile your code using the `javac` compiler. To enable the `assert` mechanism you will have to compile the `PeopleManager` class using the following `javac` command-line arguments:

```
javac -source 1.4 PeopleManager.java
```

To test the `PeopleManager` class with its new assertion capabilities you will have to enable assertions in the Java virtual machine by using the `-ea` command-line argument when you run the `PeopleManagerApplication` program. Here's how the java command would look:

```
java -ea PeopleManagerApplication
```

Example 9.18 gives the source code for the `PeopleManagerApplication` class that tests the newly added `PeopleManager` class functionality. Figure 9-11 shows the results of running this program.

*9.18 PeopleManagerApplication.java
(Testing new PeopleManager class functionality)*

```

1      public class PeopleManagerApplication {
2          public static void main(String[] args){
3              PeopleManager pm = new PeopleManager(); // default constructor test
4
5              pm.addPerson("Steven", "Jay", "Jones", 1946, 8, 30, Person.MALE);
6              pm.addPerson("Jeanie", "Sue", "Freeman", 1963, 10, 10, Person.FEMALE);
7              pm.addPerson("Richard", "Warren", "Miller", 1723, 2, 29, Person.MALE);
8
9              pm.listPeople();
10
11             pm.deletePersonAtIndex(1);
12             pm.insertPersonAtIndex(1, "Coralie", "Sylvia", "Miller", 1965, 8, 3, Person.FEMALE);
13

```

```

14     System.out.println();
15     pm.listPeople();
16
17     pm.deletePersonAtIndex(-3); // test assertion - code should fail here!
18 } // end main
19 } // end PeopleManagerApplication class

```

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/person] swodog% java -ea PeopleManagerApplication
Steven Jay Jones 58
Jeanie Sue Freeman 41
Richard Warren Miller 281

Steven Jay Jones 58
Coralie Sylvia Miller 39
Richard Warren Miller 281
Exception in thread "main" java.lang.AssertionError
    at PeopleManager.deletePersonAtIndex(PeopleManager.java:23)
    at PeopleManagerApplication.main(PeopleManagerApplication.java:17)
[Rick-Millers-Computer:~/desktop/person] swodog%

```

Figure 9-11: Results of Running Example 9.18

The assertion mechanism made the code fail as expected. Assertions are best used during code development to help with testing and debugging. Assertions are then usually disabled after the code has been thoroughly tested.

Quick Review

The `PeopleManager` class implementation process followed the same pattern as that of class `Person`. Start with the class shell and add fields and methods as required to implement the necessary functionality. Develop incrementally by applying the development cycle in an iterative fashion.

The Java assertion mechanism can be used to thoroughly test and debug code. Assertions are usually disabled upon completion of the testing phase.

MORE ABOUT METHODS

In this section I'd like to focus your attention on two behavioral aspects of methods which you will find helpful to fully understand before attempting more complex programming projects: 1) how arguments are passed to methods and, 2) local variable scoping rules.

HOW ARGUMENTS ARE PASSED TO METHODS

Two sorts of things can be passed as arguments to a method: 1) a primitive type object or 2) a reference that points to an object or array of objects. (*Otherwise simply referred to as a reference.*) When the argument is passed to the method a copy of the object is made and assigned to its associated method parameter. This is referred to as *pass by copy* or *pass by value*.

Consider for a moment the following method declaration:

```

public void someMethod(int int_param, Object object_ref_param){
    // body statements omitted
}

```

The `someMethod()` method declares two parameters: one primitive type `int` parameter and one `Object` reference parameter. This means that `someMethod()` can take two arguments: the first must be an integer primitive and the second can be a reference to any `Object`. (*Remember — all user-defined types are ultimately Objects! Also remember that a reference contains a value that represents the memory location of the object to which it points.*) The values contained in these two arguments (*an int and a reference*) are copied to their corresponding parameters during the early

stages of the call to `someMethod()`. While `someMethod()` executes it is only operating on its parameters, meaning it is only operating on copies of the original argument values.

For primitive types this simply means that any change of value made to a method's parameter will only affect the copy — not the original value. The same holds true for reference parameters. A reference parameter will point to the same object the reference argument points to unless, during the method call, the reference parameter is changed to point to a different object. This change will only affect the parameter or copy — not the original reference used as an argument to the method call. (*Bear in mind, however, that while a reference parameter points to the same object the argument points to, changes to the object made via the parameter will have the same effect as though they were made via the argument itself.*) Figure 9-12 illustrates these concepts.

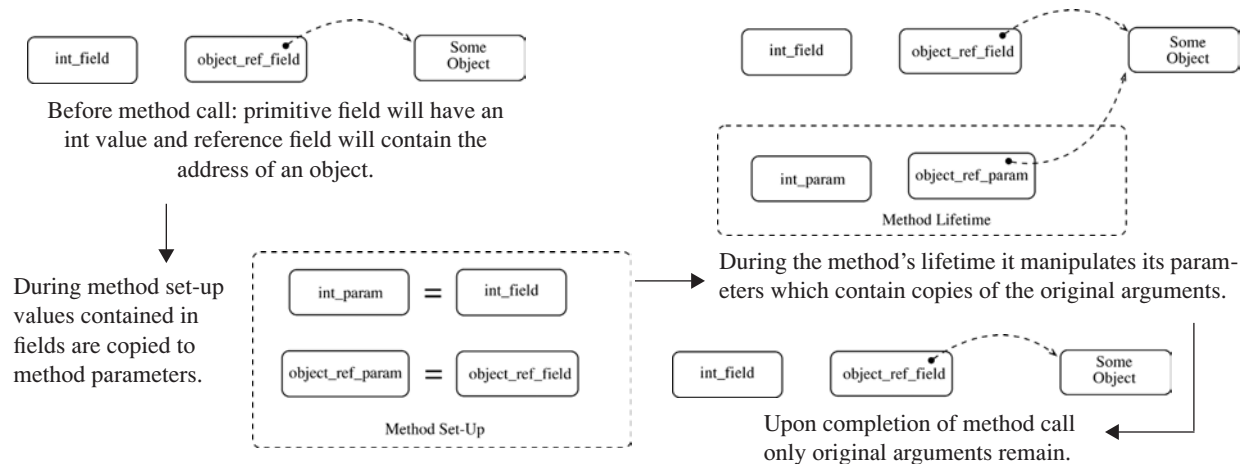


Figure 9-12: Primitive and Reference Argument Values are Copied to Method Parameters

Referring to figure 9-12 — Prior to a method call primitive and reference fields contain values. During method set-up these values are copied to their corresponding method parameters. The parameters can be manipulated by the method during the method's lifetime. Changes to the parameter values will only affect the parameters, not the original arguments. After the method call primitives and references used as arguments will have their original values. Changes to the object pointed to by the reference parameter will remain in effect.

LOCAL VARIABLE SCOPING

Methods can declare variables for use within the method body. These variables are known as *local variables*. The scope of a local variable includes the method body block or code block in which it is declared, however, it is only available for use after its point of declaration. Parameters are considered to be local variables and are available for use from the beginning to the end of the method body.

A local variable whose name is the same as a class or instance field will hide that field from the method body. To access the field you must preface its name with the `this` keyword. (*Or change the field's name or the local variable's name to eliminate the problem!*)

ANYWHERE AN OBJECT OF <TYPE> IS REQUIRED, A METHOD THAT RETURNS <TYPE> CAN BE USED

The title to this section says it all. Anywhere an object of a certain *type* is required, a method that returns a result of that *type* can be used. Substitute the word *type* in the previous sentence for any primitive or reference type you require. For reference types a `new` clause can be used as well. Refer to the following method declaration once again:

```
public void someMethod(int int_param, Object object_ref_param) {
    // body statements omitted
}
```


Assume for this example that the following fields and methods exist as well: `int_field`, `object_reference_field`, `getInt()` and `getObject()`. Assume for this example that `getInt()` returns a primitive `int` value and that `getObject()` returns a reference to an `Object`. Given these fields and methods the `someMethod()` could be called in the following ways:

```
someMethod(int_field, object_reference_field);
someMethod(getInt(), object_reference_field);
someMethod(int_field, getObject());
someMethod(getInt(), getObject());
someMethod(getInt(), new Object());
```

As you progress through this book and your knowledge of Java grows you will be exposed to all the above forms of a method call plus several more.

Quick Review

Arguments are passed to a method call by value. This is also referred to as pass by copy. The method parameters contain a copy of the argument values and any change to the parameter values only affect the copies, not the actual arguments. Any change to an object pointed to by a method parameter will remain in effect when the method call completes.

Methods can contain local variables whose scope is the body code block or the code block in which they are declared. Local variables are available for use after the point of their declaration up to the end of the code block. Method parameters are local variables that are available to the entire method body.

Anywhere an object of `<type>` is required a method that returns that `<type>` can be used in its place.

Special Topics

Before bringing this chapter to a close I'd like to talk briefly about static initializers and data encapsulation.

Static Initializers

When classes contain static fields (*a.k.a. class fields*) they must be initialized before any instance of the class is created. If you initialized the static field in the constructor then each time a new object is created the static field's value would be overwritten. This is generally not the desired behavior.

If the static field just needs to be initialized to a simple value then its declaration within the class and same-line initialization will suffice. However, if you have to perform a non-trivial initialization, like the initialization of an array, then you can use a static initializer to perform the initialization. The code in example 9.19 offers an example of a static initializer in action. Figure 9-13 shows the results of running this program.

9.19 *StaticInitializerTest.java*

```
1      public class StaticInitializerTest {
2          private static int static_int_val;
3          private static int[] static_int_array = new int[10];
4          static {
5              for(int i = 0; i < static_int_array.length; i++){
6                  static_int_array[i] = i;
7              }
8          } //end static class initializer
9
10         /***** Constructor Method *****/
11         public StaticInitializerTest(){
12             static_int_val++;
13         }
14
15         /**** printStaticArrayValues method *****/
16         public void printStaticArrayValues(){
17             for(int i = 0; i < static_int_array.length; i++){
18                 System.out.print(static_int_array[i] + " ");
19             }
20             System.out.println();
21         }
22     }
```

```

23     /***** getStaticIntVal method *****/
24     public int getStaticIntVal(){ return static_int_val; }
25
26     /**** main method *****/
27     public static void main(String[] args){
28         StaticInitializerTest sit1 = new StaticInitializerTest();
29         System.out.println(sit1.getStaticIntVal());
30
31         StaticInitializerTest sit2 = new StaticInitializerTest();
32         System.out.println(sit2.getStaticIntVal());
33
34         StaticInitializerTest sit3 = new StaticInitializerTest();
35         System.out.println(sit3.getStaticIntVal());
36
37         StaticInitializerTest sit4 = new StaticInitializerTest();
38         System.out.println(sit4.getStaticIntVal());
39
40         sit1.printStaticArrayValues();
41         sit2.printStaticArrayValues();
42         sit3.printStaticArrayValues();
43         sit4.printStaticArrayValues();
44     } //end main
45 } //end class

```

```

Terminal — tcsh
[rick-millers-computer:~/desktop/StaticInitializerTest] swodog% java StaticInitializerTest
1
2
3
4
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
[rick-millers-computer:~/desktop/StaticInitializerTest] swodog%

```

Figure 9-13: Results of Running Example 9.19

Referring to example 9.19 — Two static fields are declared for the `StaticInitializerTest` class. The `static_int_val` field is just a simple field and it is automatically initialized to zero when the class is loaded into the Java virtual machine. The `static_int_array` is a special case. When it is created on line three all the elements will be initialized to zero which is what you expect. However, if you did not want all the elements to remain at zero then you have two options: 1) use array literals to initialize the array (*see chapter 8*) or, 2) use the static initializer as is shown on line 4.

A static initializer is simply a block of code preceded by the keyword *static*. There can be more than one static initializer and they can appear anywhere within the class definition that a field or method declaration can appear, but, if they are used at all, there is usually just one and it is placed just above the first constructor method for clarity.

Static initializers behave like static methods in that they cannot use the *this* keyword and cannot reference instance variables or methods.

DATA ENCAPSULATION - THE NAKED TRUTH

You were introduced to data encapsulation early in the chapter when talking about private instance fields. I then proceeded to show you how to write getter and setter methods that manipulate a few of the private fields of the `Person` class. Getter methods, if used improperly, can violate your best efforts at encapsulation. In the case of the `Person` class, the use of getter methods to return a reference to the fields `first_name`, `middle_name`, and `last_name` was not necessarily a bad thing. This is because `String` objects are immutable. That is, `Strings` cannot be changed once created. Any `String` method such as `toUpperCase()` or `toLowerCase()` that appears to modify the `String` object actually returns a new `String` object. The immutable nature of `Strings` prevents a `Person` object's `String` fields from being modified even if a reference to them is returned.

If, however, a class contains fields of mutable (changeable) types then you must not return references to those fields willy-nilly. The requirements of your design will dictate how you enforce data encapsulation.

The idea behind data encapsulation is simply the separation of *what a class can do* from *how it actually does it*. If you mindlessly expose the innards of a class to the outside world you will have difficulty in the future should you decide to change the class's innards.

Quick Review

Static initializers can be used to perform complex initialization of static class fields. There can be more than one static initializer in a class and they can appear anywhere a field or method declaration appears.

Data encapsulation can be unwittingly violated through the improper use of getter and setter methods. Let good class design be your guide in choosing exactly what, if any, internal pieces of a class to expose.

SUMMARY

Problem abstraction requires lots of programmer creativity and represents the *Art* in the *Art of Programming*. Your guiding mantra during problem abstraction is to *Amplify the Essential — Eliminate the Irrelevant*. Problem abstraction is performed in the analysis and design phase of the development cycle. The abstractions you choose to model a particular problem will directly influence a program's design.

The end result of problem abstraction is the identification and creation of one or more new data types. The data types derived through problem abstraction are referred to as abstract data types (ADTs) or user-defined data types. User-defined data types are implemented as Java classes. These classes will interact with each other in some capacity to implement the complete problem solution.

A Unified Modeling Language (UML) class diagram is used to show the static relationship between classes that participate in a software design. The class diagram is used to by programmers to express and clarify design concepts to themselves, to other programmers, to management, and to clients.

A class is represented in UML by a rectangle. The rectangle can have three compartments. The uppermost compartment contains the class name, the middle compartment contains fields, and the bottom compartment contains the methods.

Generalization and specialization is indicated with lines tipped with hollow arrows. The arrow points from the specialized class to the generalized class. The generalized class is the base class and the specialized class is the derived or subclass. Generalizations specify “is a...” relationships between base and subclasses.

Dependencies are indicated by dashed arrows pointing to the class being depended upon. Dependencies are one way to indicate “uses...” relationships between classes.

Java classes have four types of members: 1) static fields, 2) non-static fields, 3) static methods, and 4) non-static methods. Static fields are shared between all class objects whereas each object has its very own copy of instance fields. Alternative names for static are class and class-wide. An alternative name for non-static is instance. Static methods can only directly manipulate static fields. Non-static methods can manipulate both static and non-static fields.

There are three access modifiers: public, private, and protected. All three access modifiers are used to control horizontal access. If no access is specified then default or package access is implied.

Horizontal access is the term used to describe how a client object uses the services of a server object. Public fields and methods published by a class are collectively referred to as its interface. Private fields and methods are said to be encapsulated within the class. Client code becomes dependent upon a class's interface. Changes to a class's interface can potentially break any client code that depends upon its interface.

Methods are named modules of executable program functionality. Methods contain program statements that, when grouped together, represent a basic level of code reuse. You access the functionality of a method by calling the method using its name in a program.

Methods should be well-named and maximally cohesive. A well-named, maximally-cohesive method will pull no surprises.

Method definitions have structure. Their behavior can be optionally modified with method modifiers, they can optionally specify a return result type or void, and they can have an optional parameter list.

Methods have a distinguishing characteristic known as a method signature. Methods with different names and parameter lists are said to have different signatures. Methods with different names and the same parameter list also have different signatures. Methods with the same name and different parameter lists have different signatures as well and are said to be overloaded (*because they share the same name*). Methods cannot have the same name and identical parameter lists. This will cause a compiler error.

Constructor methods are used to set up or build an object when it's created in memory. Java will provide a default constructor but it may or may not provide the level of functionality required.

Abstract data types can be incrementally built and tested by iteratively applying the steps of the development cycle. Start with the class definition shell and then add fields and methods as required to fulfill the class's design objectives.

Class functionality can be testing with the help of a test driver. A test driver is a small program that's used to exercise the functionality of another program.

The Java assertion mechanism can be used to thoroughly test and debug code. Assertions are usually disabled upon completion of the testing phase.

Argument values are passed to methods by value. This is also referred to as pass by copy. The method parameters contain a copy of the argument values and any change to the parameter values only affect the copies, not the actual arguments. Any change to an object pointed to by a method parameter will remain in effect when the method call completes.

Methods can contain local variables whose scope is the body code block or the code block in which they are declared. Local variables are available for use after the point of their declaration up to the end of the code block. Method parameters are local variables that are available to the entire method body.

Anywhere an object of <type> is required a method that returns that <type> can be used in its place.

Static initializers can be used to perform complex initialization of static class fields. There can be more than one static initializer in a class and they can appear anywhere a field or method declaration can appear.

Data encapsulation can be unwittingly violated through the improper use of getter and setter methods. Let good class design be your guide in choosing exactly what, if any, internal pieces of a class to expose.

Skill-Building Exercises

1. **Java API Drill:** Browse through the Java API and look for classes that contain static class methods or fields. Note how they are being used in each class.
2. **Problem Abstraction Drill:** Revisit the robot rat project presented in chapter 3. Study the project specification and identify candidate classes. Make a table of the classes and list their names along with a description of their potential fields and functionality. Try not to be influenced by the solution approach taken in chapter 3. Instead, focus on breaking the problem into potential classes and assigning functionality to those classes. For example, the program written in chapter 3 is included in one large application class. At a minimum you will want to have a separate application class. Draw a UML diagram to express your design.
3. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of an automobile in code. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.
4. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of an airplane in code. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.
5. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of a nuclear submarine. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.
6. **Further Research:** Research the topic of data encapsulation. The goal of your research should be to understand the role design plays in determining the level of data encapsulation and what design and programming strategies you can use to enforce data encapsulation.
7. **Coding Exercise:** Write a program that lets you experiment with the effects of method parameter passing. The names of the class and any fields and methods required are left to your discretion. The idea is to create a class that contains several primitive and reference fields. It should also contain several methods that return primitive types and references. Write a method that takes at least one primitive type and one reference type parameter. Practice

calling the method using a combination of fields, methods, and the new operator. Manipulate the parameters in the body of the method and note the results.

8. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of a computer in code. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.
9. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of a coffee maker. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.
10. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of a gas pump. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.

SUGGESTED PROJECTS

1. **Improve The PeopleManager.addPerson() Method:** Improve the functionality of the addPerson() method of the PeopleManager class presented in this chapter. In its current state the addPerson() method only creates a new Person object and assigns the reference to the array element if the array element is null. Otherwise it does nothing and gives no indication that the creation and insertion of a new Person object failed. Make the following modifications to the addPerson() method:
 - a. Search the people_array for a null element and insert the new person at that element.
 - b. If the array is full create a new array that's 1.5 times the size of the current people_array and then insert the new Person reference at that element.
 - c. Have the addPerson() method return a boolean value indicating success or failure of the new Person object creation and insertion operation.
2. **Write A Submarine Commander Program:** Using the results of the problem abstraction performed in skill-building exercise 5 write a program that lets you create a fleet of nuclear submarines. You should be able to add submarines to the fleet, remove them from the fleet, and list all the submarines in your fleet. You will want to power-up their nuclear reactors and shut down their nuclear reactors. You will also want to fire their weapons. To keep this programming exercise manageable just write simple messages to the console in response to commands sent to each submarine object.
3. **Write A Gas Pump Operation Program:** Using the results of the analysis you performed in skill-building exercise 10 write a program that lets you control the operation of a gas pump. You should be able to turn the gas pump on and off. You should only be able to pump gas when the pump is on. When you are done pumping gas indicate how much gas was pumped in gallons or liters and give the total price of the gas pumped. Provide a way to set the price of gas.
4. **Write A Calculator Program:** Write a program that implements the functionality of a simple calculator. The focus of this project should be the creation of a class that performs the calculator's operations. Give the Calculator class the ability to add, subtract, multiply, and divide integers and floating point numbers. Some of the Calculator class methods may need to be overloaded to handle different types of arguments.
5. **Write A Library Manager Program:** Write a program that lets you catalog the books in your personal library. The Book class should have the following attributes: title, author, and ISBN. You can add any other attributes you deem necessary. Create a class named LibraryManager that can create and add books to your library, delete books from your library, and list the book in your library. Use an array to hold the books in your library. Research sorting routines and implement a sortBooks() method.
6. **Write A Linked-List Program:** A special property of Java classes is that the name of the class you are defining

can be used to declare fields within that class. Consider the following code example:

```

1     public class Node {
2         private Node previous = null;
3         private Node next = null;
4         private Object payload = null;
5
6         // methods omitted for now
7     }

```

9.20 Node.java (Partial Listing)

Here, the class name Node appears in the body of the Node class definition to declare two Node references named previous and next. This technique is used to create data structures designed for use within a linked list. Use the code shown in example 9.20 to help you write a program that manages a linked list. Here are a few hints to get you started:

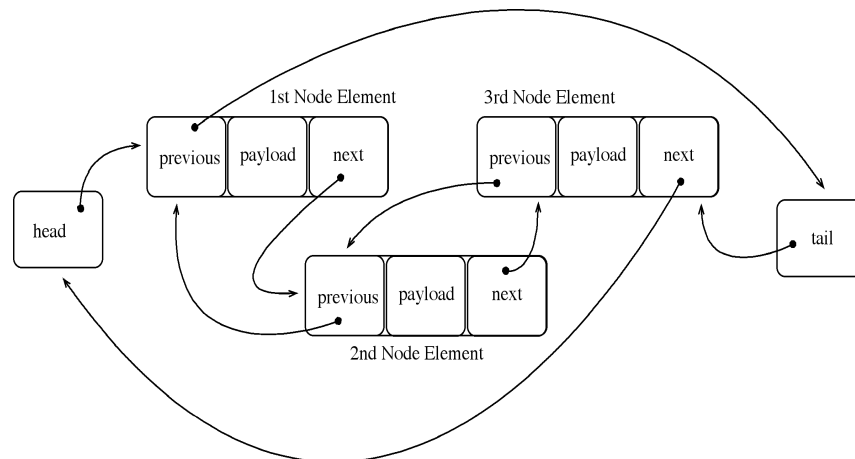


Figure 9-14: Linked List with Three Nodes

Referring to figure 9-14 above — a linked list is comprised of a number of nodes and a head and a tail. The head points to the first node in the list. The tail always points to the last node in the list. Each node has a next and previous attribute along with a payload. Figure 9-14 shows a linked list having three nodes. The first node element's next attribute points to the second node element, and the second node element's next attribute points to the third node element. The third node element is the last node in the list and its next attribute points to the head, which always points to the first node in the list. Each node's previous attribute works in the opposite fashion.

Because each node has a next and a previous attribute they can be used to create circular linked list as is shown in figure 9-14.

For this project write a linked list manager program that lets you add, delete, and list the contents of each node in the list. You will have to add methods to the Node class code given in example 9.20. At a minimum you should add getter and setter methods for each field.

This is a challenging project and will require you to put some thought into the design of both the Node and the LinkedListManager class. The most complicated part of the design will be figuring out how to insert and delete nodes into and from the list. When you successfully complete this project you will have a good, practical understanding of references and how they are related to pointers in other programming languages.

7. Convert The Library Manager To Use A Linked List: Rewrite the library manager program presented in suggested project 6 to use a linked list of books instead of an array.

SELF-TEST QUESTIONS

1. Define the term problem abstraction. Explain why problem abstraction requires creativity.

2. What is the end result of problem abstraction?
3. Describe in your own words how you would go about the problem abstraction process for a typical programming problem.
4. What is the purpose of the UML class diagram? What geometric shape is used to depict classes in a UML class diagram? Where are class names, fields, and methods depicted on a class symbol?
5. What do the lines tipped with hollow arrowheads depict in a UML class diagram?
6. List the four categories of Java class members.
7. Explain the difference between static and non-static fields.
8. Explain the difference between static and non-static methods.
9. What are the alternative names used to denote static and non-static fields and methods?
10. List and describe the three access modifiers.
11. Explain the concept of horizontal access. Draw a picture showing how client code access to a server class's fields and methods is controlled using the access modifiers public and private.
12. What is the purpose of data encapsulation? How can data encapsulation be unwittingly violated?
13. What is a method?
14. List and describe the desirable characteristics of a method.
15. Explain the concept of cohesion.
16. (True/False) A method should be maximally cohesive.
17. What steps can you take as a programmer to ensure your methods are maximally cohesive?
18. What's the purpose of a method definition?
19. What parts of a method definition are optional?
20. What is meant by the term method signature?
21. What parts of a method are included in a method's signature?
22. What constitutes an overloaded method?
23. Give at least one example of where method overloading is useful.
24. What makes constructor methods different from ordinary methods?
25. Describe in your own words how arguments are passed to methods.
26. (True/False) In programming situations that call for an object of a certain type, methods that return that same type can be used.

27. What's the purpose of a static initializer?
28. How many static initializers can there be in a class definition?

REFERENCES

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*. O'Reilly and Associates, Inc. Sebastopol, CA. ISBN: 0-596-00283-1

James Gosling, et. al. *The Java™ Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, MA. ISBN: 0-201-31008-2

Jon Meyer, et. al. *Java™ Virtual Machine*. O'Reilly and Associates, Inc., Sebastopol, CA. ISBN: 1-56592-194-1

Rick Miller. *C++ For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-02-8

Donald E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Second Edition. Addison-Wesley. Reading MA. ISBN: 0-201-89685-0

Grady Booch. *Object-Oriented Analysis And Design With Applications*, Second Edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA. ISBN: 0-8053-5340-2

Sinan Si Alhir. *UML In A Nutshell: A Desktop Quick Reference*. O'Reilly and Associates, Inc., Sebastopol, CA. ISBN: 1-56592-448-7

NOTES

CHAPTER 10



STREET PIG

COMPOSITIONAL DESIGN

LEARNING OBJECTIVES

- STATE THE DEFINITION OF SIMPLE AGGREGATION
- STATE THE DEFINITION OF COMPOSITE AGGREGATION
- EXPLAIN THE DIFFERENCE BETWEEN SIMPLE AND COMPOSITE AGGREGATION
- EXPRESS SIMPLE AND COMPOSITE AGGREGATION USING A UML CLASS DIAGRAM
- EXPRESS AN ASSOCIATION BETWEEN TWO CLASS TYPES USING A UML CLASS DIAGRAM
- DEFINE THE TERMS HAS A, CONTAINS, & USES IN THE CONTEXT OF COMPOSITIONAL DESIGN
- EXPLAIN THE DIFFERENCE BETWEEN A DEPENDENCY AND AN ASSOCIATION
- EXPLAIN THE CLIENT/SERVER RELATIONSHIP BETWEEN A CONTAINING AND A CONTAINED CLASS
- DEMONSTRATE YOUR ABILITY TO USE SIMPLE AND COMPOSITE AGGREGATION IN YOUR PROGRAM DESIGN
- EXPLAIN HOW TO IMPLEMENT MESSAGE PASSING BETWEEN OBJECTS
- STATE THE PURPOSE AND USE OF A UML SEQUENCE DIAGRAM

INTRODUCTION

Rarely is an application comprised of just one class. In reality, applications are constructed from many classes, each providing a unique service. This chapter introduces you to the concepts and terminology associated with building complex application behavior from a collection of classes. This is referred to as *compositional design* or *design by composition*.

The study of compositional design is the study of *aggregation* and *containment*. In this chapter you will learn the two primary aggregation associations: *simple* and *composite*. You will also learn how to use a UML class diagram to illustrate the static relationship between classes in a complex application. To do this you will need to know how to express simple and composite aggregation visually.

The study of aggregation also entails learning how a *whole* class accesses the services of its *part* classes. The concepts of *message passing* and *sequencing* will be demonstrated by introducing you to a new type of UML diagram known as the sequence diagram.

MANAGING CONCEPTUAL AND PHYSICAL COMPLEXITY

Programs that depend upon the services of multiple classes are inherently more complex than those that do not. This complexity takes two forms: 1) the *conceptual complexity* derived from the nature of the relationship between or among classes that participate in the design, and 2) the *physical complexity* that results from having to deal with multiple source files. In this chapter you will encounter programming examples that are both conceptually and physically more complex than previous examples.

You will manage a program's conceptual complexity by using object-oriented design principles in conjunction with the Unified Modeling Language (UML) to express a program's design. In this chapter you will learn the concepts of building complex programs using compositional design. You will also learn how to express compositional design using the UML.

The physical complexity of the programs you write in this chapter must be managed through source-file organization. If you have not started to put related project source files in one folder you will want to start doing so. This will make them easier to manage — at least for the purposes of this book. You will also need to change the way you compile your project source files. Until now I have shown you how to compile source files one at a time. However, when you are working on projects that contain many source files, these files may have dependencies to other source files in the project. A change to one source file will require a recompilation of all the other source files that depend upon it.

If you use a Java integrated development environment (IDE) it will manage the source file dependencies automatically. If you are using Sun's Java Software Development Kit (SDK) you can use an open source build tool like Ant to manage your project's build process. Ant is used by professional Java developers to manage large project builds. It would be a good idea to learn how to use Ant although a discussion of this tool is beyond the scope of this book. However, if you are interested in Ant I recommend visiting the Ant website. [<http://ant.apache.org/>] I have also listed a good Ant book in the references section.

But wait! You don't need to learn Ant right this minute. Read the next section first.

COMPILING MULTIPLE SOURCE FILES SIMULTANEOUSLY WITH JAVAC

If you don't have time to learn Ant and don't have a fancy schmancy IDE don't despair. You can compile multiple Java source files simultaneously using the javac compiler tool. All you need to do is enter the names of the source files you want to compile on the command line following the javac command as is shown in the following example:

```
javac SourceFileOne.java SourceFileTwo.java SourceFileThree.java
```

Each source file is listed on the command line one after the other separated by a space. You can compile as many source files as you require using this method.

If you are working on a project that has more than two or three source files then you may want to create a batch file (*Windows*) or a shell script (*Linux, Macintosh OS X*) that you can call to perform the compilation for you.

Another way to compile multiple related source files is to put them in a project directory (*Which you should be doing anyway!*) and use the `javac` compiler tool as is shown in the following example:

```
javac *.java
```

This will compile all the source files in a given directory. You may, of course, provide an explicit directory path as shown in the following example:

```
javac c:\myprojects\project1\*.java
```

Quick Review

There are two types of complexity: conceptual and physical. Conceptual complexity is managed by using object-oriented design concepts and expressing your object-oriented designs in the Unified Modeling Language (UML). Physical complexity can be managed by your IDE or with an automated build tool such as Ant. You can also manage physical complexity on a small scale by organizing your source files into project directories and compiling multiple files simultaneously using the `javac` compiler tool.

Dependency vs. Association

A Java program built from many classes will manifest several types of interclass relationships. Before continuing with this chapter you must be clear in your understanding of the terms *dependency* and *association*.

A dependency is a relationship between two classes in which a change made to one class will have an affect on the behavior of the class or classes that depend on it. For example, if you have two classes, class A and class B, and class A depends upon the behavior of class B in a way that a change to class B will affect class A, then class A has a dependency on class B. If you use a class in your program and write code that calls one or more of that class's interface methods then your code is dependent upon that class. If its interface methods change your code might break. If the behavior of those methods change your program's behavior will change as well.

All but the most trivial Java programs you write will be chock full of dependency relationships between your classes and, at the very least, the classes you use in your programs that are supplied by the Java Platform API.

An association is a relationship between two classes that denotes a connection between those classes. An association implies a peer-to-peer relationship between the classes that participate in the association. If you have two classes, class A and class B, and there is an association between class A and class B, then class A and class B are linked together at the same level of importance. They may each depend on the other and the link between them will be navigable in one, or perhaps two, directions.

An aggregation is a special type of association and is discussed in detail in the following section.

Aggregation

An *aggregation* is an *association* between two objects that results in a whole/part relationship. An object comprised of other objects (*a.k.a. the whole object*) is referred to as an *aggregate*. Objects used to build the whole object are called *part objects*. There are two types of aggregation: *simple* and *composite*. Each type of aggregation is discussed in detail below.

Simple vs. Composite Aggregation

Aggregate objects can be built from one or more part objects. An aggregate object can be a simple aggregate, a composite aggregate, or a combination of both. The difference between simple and composite aggregation can be found in how the whole or aggregate object is linked to its part objects. The type of linking between an aggregate and its parts will dictate who controls the lifetime (*creation and destruction*) of its part objects.

The Relationship Between Aggregation and Object Lifetime

The difference between simple and composite aggregation is dictated by who (i.e., what object) controls the lifetime of the aggregate's part objects.

Simple Aggregation

If the aggregate object simply uses its part objects and otherwise has no control over their creation or existence then the relationship between the aggregate and its parts is a simple aggregation. The part object can exist (*i.e. it can be created and destroyed*) outside the lifetime of the simple aggregate object. This leads to the possibility that a part object can participate, perhaps simultaneously, in more than one simple aggregation relationship.

The Java garbage collector will eventually destroy unreferenced objects, but so long as the simple aggregate object maintains a reference to its part object, the part object will be maintained in memory.

References to existing part objects can be passed to aggregate object constructors or other aggregate object methods as dictated by program design. This type of aggregation is also called *Containment By Reference*.

Composite Aggregation

If the aggregate object controls the lifetime of its part objects (*i.e., it creates and destroys them*) then it is a composite aggregate. Composite aggregates have complete control over the existence of their part objects. This means that a composite aggregate's part objects do not come into existence until the aggregate object is created.

Composite aggregate part objects are created when the aggregate object is created. Aggregate part creation usually takes place in the aggregate object constructor. However, in Java, there is no direct way for a programmer to destroy an object. You must rely on the JVM garbage collector to collect unreferenced objects. Therefore, during an aggregate object's lifetime it must guard against violating data encapsulation by not returning a reference to its part objects. Strict enforcement of this rule will largely be dictated by program design objectives.

This type of aggregation is also called *Containment By Value*.

Quick Review

An aggregation is an association between two objects that results in a whole/part relationship. There are two types of aggregation: simple and composite. The type of aggregation is determined by the extent to which the whole object controls the lifetime of its part objects. If the whole object simply uses the services of a part object but does not control its lifetime then it's a simple aggregation. On the other hand, if the whole object creates and controls the lifetime of its part objects then it is a composite aggregate.

Expressing Aggregation In A UML Class Diagram

The UML class diagram can be used to show aggregate associations between classes. This section shows you how to use the UML class diagram to express simple and composite aggregation.

Simple Aggregation

Figure 10-1 shows a UML diagram expressing simple aggregation between classes Whole and Part. The association is expressed via the line that links Whole and Part. The simple aggregation property is denoted by the hollow dia-

mond shape used to anchor the line to the Whole class. Simple aggregation represents a *uses* relationship between the aggregate and its parts since the lifetime of part objects are not controlled by the aggregate.

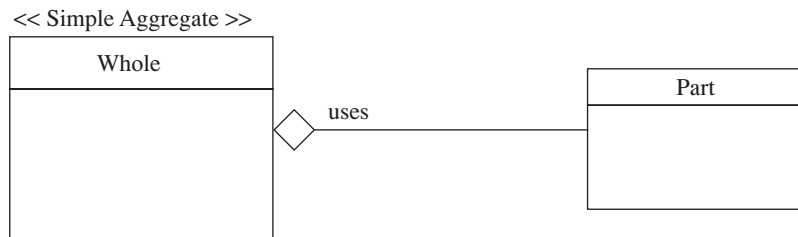


Figure 10-1: UML Diagram Showing Simple Aggregation

Figure 10-2 shows two different simple aggregate classes, Whole A and Whole B, sharing an instance of the Part class. Such a sharing situation may require thread synchronization. (See chapter 16 - Threads)

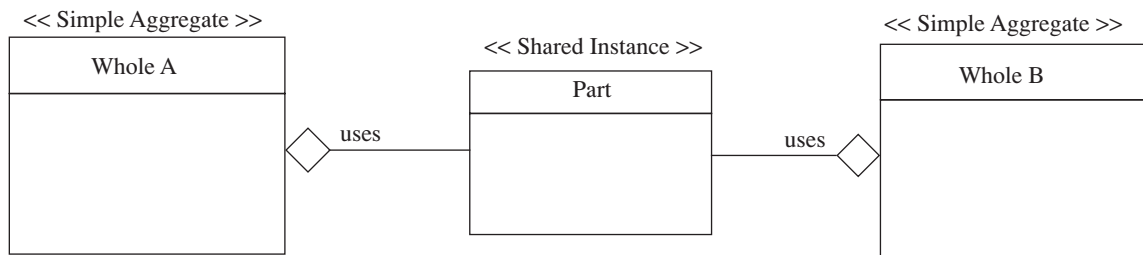


Figure 10-2: Part Class Shared Between Simple Aggregate Classes

COMPOSITE AGGREGATION

Composite aggregation is denoted by a solid diamond adorning the side of the aggregate class. Composite aggregate objects control the creation of their part objects which means part objects belong fully to their containing aggregate. Thus, a composite aggregation denotes a *contains* or *has a* relationship between whole and part classes. Figure 10-3 shows a UML diagram expressing composite aggregation between classes Whole and Part.

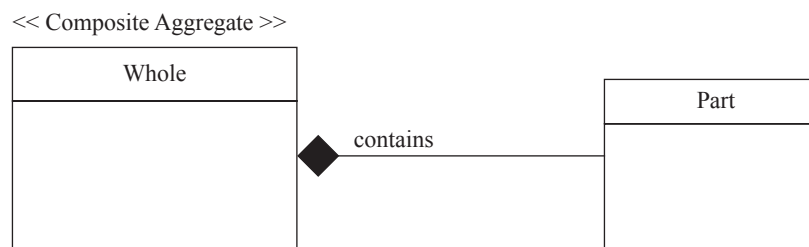


Figure 10-3: UML Diagram Showing Composite Aggregation

AGGREGATION EXAMPLE CODE

At this point it will prove helpful to you to see a few short example programs that implement simple and composite aggregation before attempting a more complex example. Let's start with a simple aggregation.

Simple Aggregation Example

Figure 10-4 gives a UML diagram showing a simple aggregate class named A that uses the services of a part class named B.

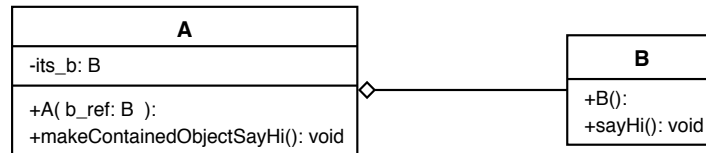


Figure 10-4: Simple Aggregation Example

Class A has one attribute, `its_b`, which is a reference to an object of class B. A reference to a B object is passed into an object of class A via a constructor argument when an object of class A is created. Class A has another method named `makeContainedObjectSayHi()` which returns `void`.

Class B has no attributes, a constructor method, and another method named `sayHi()`. Examples 10.1 and 10.2 give the code for these two classes.

10.1 A.java

```

1      public class A {
2          private B its_b = null;
3          public A(B b_ref){
4              its_b = b_ref;
5              System.out.println("A object created!");
6          }
7
8          public void makeContainedObjectSayHi(){
9              its_b.sayHi();
10         }
11     }
  
```

Referring to example 10.1 — the `its_b` reference variable is declared on line 2 and initialized to `null`. The constructor takes a reference to a B object and assigns it to the `its_b` variable. The `its_b` variable is then used on line 9 in the `makeContainedObjectSayHi()` method to call its `sayHi()` method.

10.2 B.java

```

1      public class B {
2          public B(){
3              System.out.println("B object created!");
4          }
5
6          public void sayHi(){
7              System.out.println("Hi!");
8          }
9      }
  
```

The only thing to note in example 10.2 is the `sayHi()` method which starts on line 6. It just prints a simple message to the console. Example 10.3 gives a short program called `TestDriver` that is used to test classes A and B and illustrate simple aggregation.

10.3 TestDriver.java

```

1      public class TestDriver {
2          public static void main(String[] args){
3              B b = new B();
4              A a = new A(b);
5              a.makeContainedObjectSayHi();
6          }
7      }
  
```

Referring to example 10.3 — a B object is first created on line 3. The reference `b` is used as an argument to the `A()` constructor. In this manner an A object gets a reference to a B object. On line 5 the reference `a` is used to call the `makeContainedObjectSayHi()` method. The results of running this program are shown in figure 10-5.

```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/classa_and_b_ver2] swodog% java TestDriver
B object created!
A object created!
Hi!
[Rick-Millers-Computer:~/desktop/classa_and_b_ver2] swodog% 
```

Figure 10-5: Results of Running Example 10.3

COMPOSITE AGGREGATION EXAMPLE

Figure 10-6 gives a UML diagram that illustrates composite aggregation between classes A and B.

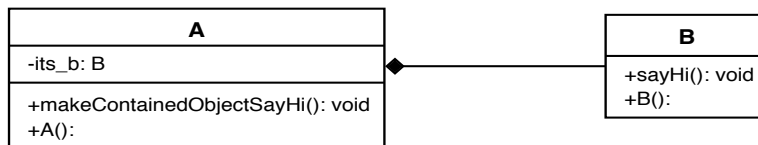


Figure 10-6: Composite Aggregation Example

Referring to figure 10-6 — class A still has an attribute named `its_b` and a method named `makeContainedObjectSayHi()`, however, the `A()` constructor has no parameters. Class B is exactly the same here as it was in the previous example. The source code for both these classes is given in examples 10.4 and 10.5.

```
1 public class A {
2     private B its_b = null;
3     public A(){
4         its_b = new B();
5         System.out.println("A object created!"); }
6
7     public void makeContainedObjectSayHi(){
8         its_b.sayHi();
9     }
10 }
```

10.4 A.java

```
1 public class B {
2     public B(){
3         System.out.println("B object created!");
4     }
5
6     public void sayHi(){
7         System.out.println("Hi!");
8     }
9 }
```

10.5 B.java

Referring to example 10.4 — the `its_b` attribute is declared and initialized to null on line 2. In the `A()` constructor on line 4 a new B object is created and its memory location is assigned to the `its_b` reference. In this manner the A object controls the creation of the B object.

Example 10.6 gives a modified version of the `TestDriver` program. Compare example 10.6 with example 10.3 above. This version is exactly one line shorter than the previous version. This is because there's no need to create a B object before creating the A object. Figure 10-7 shows the results of running this program.

```
1 public class TestDriver {
2     public static void main(String[] args){
3         A a = new A();
4         a.makeContainedObjectSayHi();
5     }
6 }
```

10.6 TestDriver.java


```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/classA_and_b] swodog% java TestDriver
B object created!
A object created!
Hi!
[Rick-Millers-Computer:~/desktop/classA_and_b] swodog%

```

Figure 10-7: Results of Running Example 10.6

Quick Review

The UML class diagram can be used to show aggregation associations between classes. A hollow diamond is used to denote simple aggregation and expresses a “uses” or “uses a” relationship between the whole and part classes. A solid diamond is used to show composite aggregation and expresses a “has” or “has a” relationship between whole and part classes.

SEQUENCE DIAGRAMS

A sequence diagram is another type of UML diagram that is used to illustrate the order or sequence of execution events that occur between objects in an application. Sequence diagrams become extremely helpful and important especially when using compositional design. Figure 10-8 shows the sequence diagram for the simple aggregation example program given in examples 10.1 through 10.3 in the previous section.

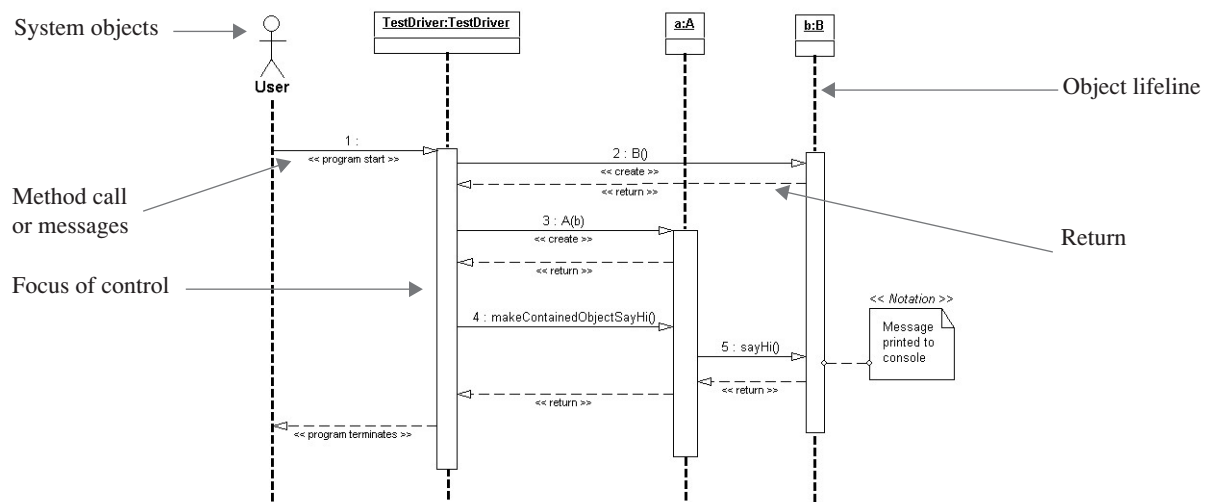


Figure 10-8: Sequence Diagram — Simple Aggregation

Referring to figure 10-8 — The sequence diagram is read from left to right. The objects that participate in the sequence of events appear along the top of the diagram. Each object has an object lifeline. An object can be an instance of a class or an external system (*actor*) that participates in the event sequence.

The User initiates the event sequence by starting the program. This is done by running the TestDriver program with the java command-line tool. The TestDriver program creates an instance of class B by calling the B() constructor method. Upon the constructor call return the TestDriver program then creates an A object by calling the A() constructor method passing the reference b as an argument. The TestDriver then sends the makeContainedObjectSayHi() message to the A object. (*i.e., a method call*) The A object in turn sends the sayHi() message to the B object. This results in the message “Hi!” being printed to the console. After the message is printed the program terminates.

The sequence of events is a little different for the composite aggregate version of this program as figure 10-9 illustrates.

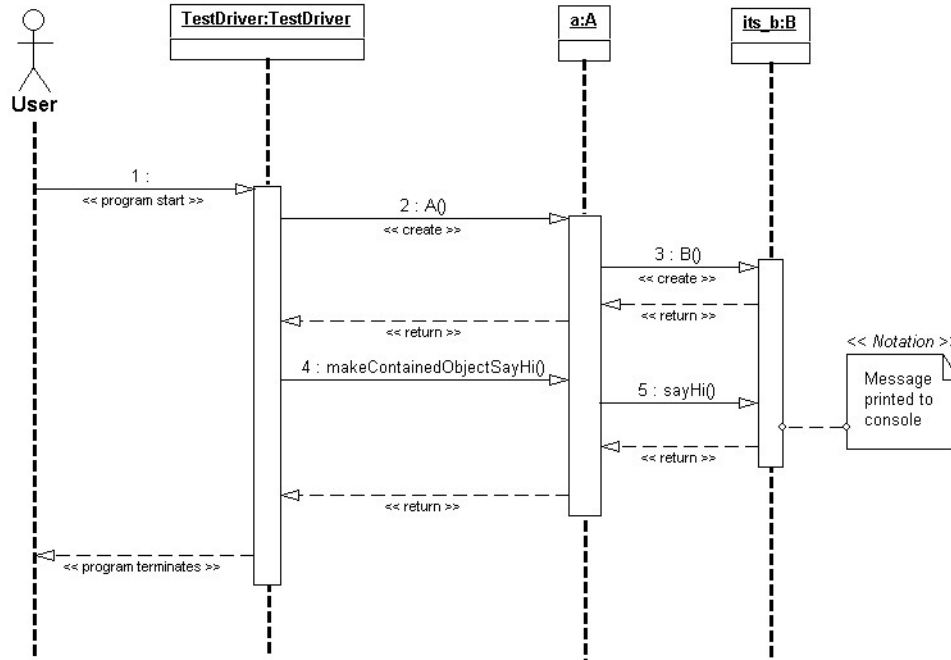


Figure 10-9: Sequence Diagram — Composite Aggregation

In the composite aggregate sequence, the TestDriver program creates the A object first. The A object then creates the B object. This is shown in message numbers 2 and 3. Messages 4 and 5 are the same here as they were in figure 10-8.

The sequence diagrams shown here are unique in that the example program is small enough to show the whole shebang in one picture. In reality, however, systems are so complex that engineers usually create sequence diagrams that focus on one piece of functionality. You’ll see an example of this in the next section.

Quick Review

Sequence diagrams are a type of UML diagram used to graphically illustrate a sequence of system events. Sequence event participants can be internal system objects or external actors. Sequence diagrams do a good job of illustrating the complex message passing between objects that participate in a compositional design.

THE AIRCRAFT ENGINE SIMULATION: AN EXTENDED EXAMPLE

In this section I will ramp up the complexity somewhat and present an extended example of compositional design — the aircraft engine simulation. Don’t panic! It’s a simple simulation designed primarily to get you comfortable with an increased level of both conceptual and physical complexity. This means there are more classes that participate in the design which means there are more source code files to create, compile, and maintain.

Figure 10-10 gives the project specification for the aircraft engine simulation. Read it carefully before proceeding to the next section.

The project specification offers good direction and several hints regarding the project requirements. The aircraft engine simulation consists of seven classes. The Engine class is the composite. An engine has a fuel pump, oil pump, compressor, oxygen sensor, and temperature sensor. These parts are represented in the design by the FuelPump,

Project Specification
Aircraft Engine Simulation

Objectives:

- Apply compositional design techniques to create a Java program
- Create a composite aggregation class whose functionality is derived from its various part classes
- Employ UML class and sequence diagrams to express your design ideas
- Derive user-defined data types to model a problem domain
- Employ inter-object message passing
- Employ the type-safe enumeration pattern in your design

Tasks:

- Write a program that simulates the basic functionality of an aircraft engine. The engine should contain the following parts: fuel pump, oil pump, compressor, temperature sensor, and oxygen sensor. Limit the functionality to the following functions:
 - * set and check each engine part status
 - * set and check the overall engine status
 - * start the engine
 - * stop the engine

Hints:

- Each engine should have an assigned engine number and each part contained by the engine should register itself with the engine number.
- When checking the engine status while the engine is running it should stop running if it detects a fault in one of its parts.
- When an individual part status changes the engine must perform a comprehensive status check on itself.
- If, when trying to start the engine, the engine detects a faulty part the engine must not start until the status on the faulty part changes.
- Limit the user interface to simple text messages printed to the console.

Figure 10-10: Aircraft Engine Project Specification

OilPump, Compressor, OxygenSensor, and TemperatureSensor classes respectively. Each of these classes have an association with the PartStatus class. The purpose of the PartStatus class is discussed in detail below.

THE PURPOSE OF THE ENGINE CLASS

The purpose of the Engine class is to embody the behavior of this thing we are modeling called an engine. A UML class diagram for the Engine class is given in figure 10-12.

ENGINE CLASS ATTRIBUTES AND METHODS

Referring to figure 10-12 — the Engine class has several attributes: *its_compressor*, *its_fuelpump*, *its_oilpump*, *its_oxygensensor*, and *its_temperaturesensor*. Each map to their respective part classes. However, several more attributes are required to complete the class. Two of these, *its_engine_number* and *is_running* are primitive type variables. The last attribute, *its_status*, is a PartStatus reference.

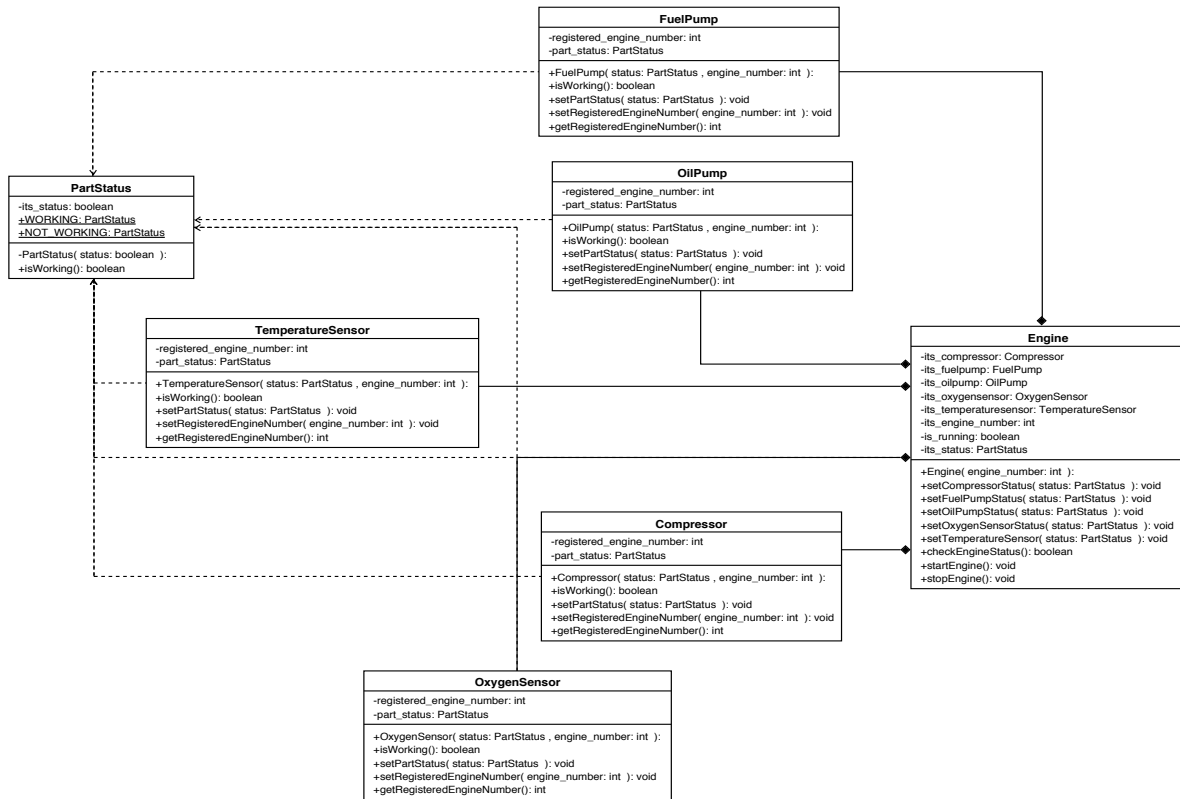


Figure 10-11: Engine Simulation Class Diagram

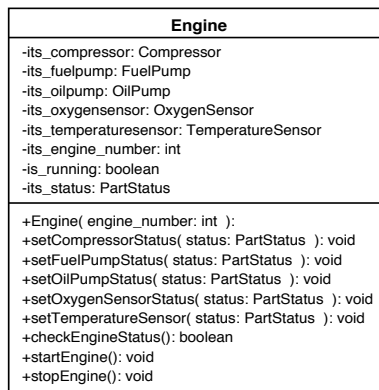


Figure 10-12: Engine Class

As you can tell from looking at the class diagram in figure 10-12 all the Engine class attributes are declared to be private. (*this is denoted by the '-' symbol preceding each attribute's name*) The design of Engine class will guard against returning a reference to any of its part objects. This will ensure that when a reference to an Engine object goes out of scope the Engine object it referenced, along with all its component objects, will be collected by the JVM garbage collector. (*Although you cannot guarantee when this collection event will occur!*)

The Engine class has one constructor that takes an integer as an argument. When created, an Engine object will set *its_engine_number* attribute using this number.

The remaining Engine class methods map pretty much directly to those specified or hinted at in the project specification. The Engine class interface allows you to set the status of each of an engine's component parts, check the status of an engine, and start and stop an engine.

PARTSTATUS —A TYPESAFE ENUMERATION PATTERN

The PartStatus class is an implementation of a popular Java design pattern known as the Typesafe Enumeration Pattern. C and C++ programmers liked the enumeration construct those languages offered but were left high-and-dry by Java and its lack of enumeration support. Although Java 1.5 now offers enumerations, early versions of Java did not.

The typesafe enumeration pattern is employed when you want to have an object assume certain valid states and have the compiler warn you when you attempt to assign an invalid state to an object.

The specific problem encountered in the aircraft engine simulation program is that of setting the part status. If you define two valid part status states in your design as being WORKING and NOT_WORKING how do you ensure that only those states are assigned to a part's status attribute?

If you attempt to solve the problem using a primitive type integer variable and define two constants like so:

```
public static final int WORKING = 0;
public static final int NOT_WORKING = 1;
```

...you can then declare an attribute named its_status and assign to it one of these constant values:

```
private int its_status = WORKING;
```

The value of its_status is now set to 0, which is a valid state value in your design, however, nothing in the Java compiler will prevent you from assigning an unacceptable integer value to its_status. Take a look now at the code for the PartStatus class given in example 10-7.

10.7 PartStatus.java

```
1     public class PartStatus {
2         private boolean its_status = false;
3
4         public static final PartStatus WORKING = new PartStatus(true);
5         public static final PartStatus NOT_WORKING = new PartStatus(false);
6
7         private PartStatus(boolean status){
8             its_status = status;
9         }
10
11        public boolean isWorking(){ return its_status; }
12    }
```

This type of programming takes a little getting used to. The PartStatus class has one private attribute which is a boolean variable. It has two public constant attributes WORKING and NOT_WORKING which are instances of type PartStatus. Notice that the PartStatus class has a private constructor. This means that the only code allowed to create instances of the PartStatus class is itself, which is done on lines 4 and 5. Study the code listing given at the end of the chapter and see how the PartStatus class and its typesafe enumeration functionality is utilized in the program.

AIRCRAFT ENGINE SIMULATION SEQUENCE DIAGRAMS

The aircraft engine simulation is sufficiently complex to warrant focused sequence diagrams. Figure 10-13 gives the sequence diagram for the creation of an Engine object.

Referring to figure 10-13 — a User starts the sequence of events by running the EngineTester program. (*The EngineTester class is covered in the next section.*) The EngineTester program creates an Engine object by calling the Engine() constructor with an integer argument. The Engine constructor in turn creates all the required part objects. When all the part object constructor calls return, the Engine constructor call returns to the EngineTester program. At this point the Engine object is ready for additional interface method calls from the EngineTester program.

As I said earlier, this sequence diagram represents a focused part of the aircraft simulation program. In fact, this diagram only accounts for the sequence of events resulting from the execution of line number 3 of example 10.8 in the next section. The creation of additional sequence diagrams for the aircraft simulation program is left for you to do as an exercise.

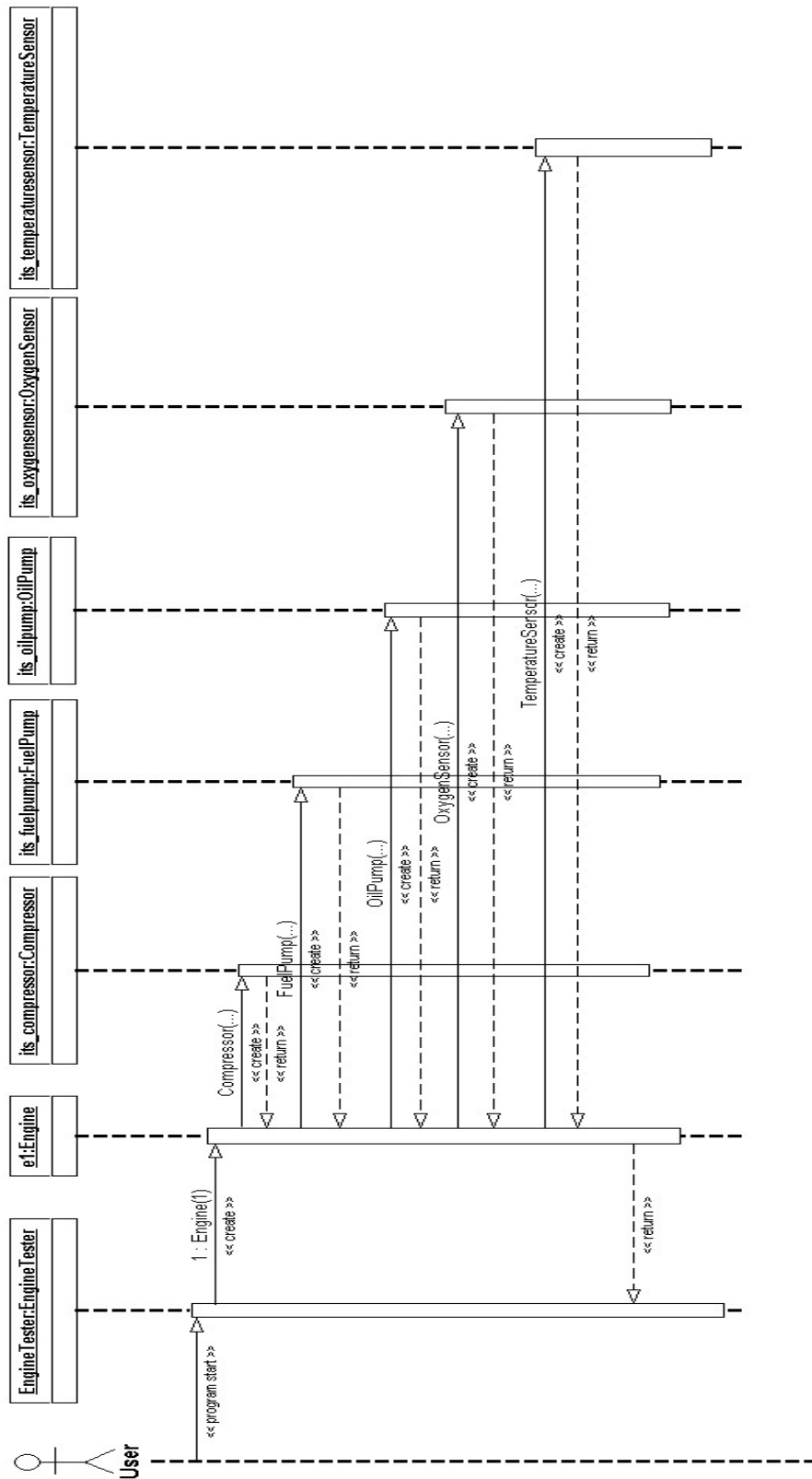


Figure 10-13: Aircraft Engine Create Engine Object Sequence

RUNNING THE AIRCRAFT ENGINE SIMULATION PROGRAM

To run the aircraft engine simulation you'll need to create a test driver program. My version of the test driver program is a class named `EngineTester` and is shown in example 10-8. The result of running this program is shown in figure 10-14.

10.8 *EngineTester.java*

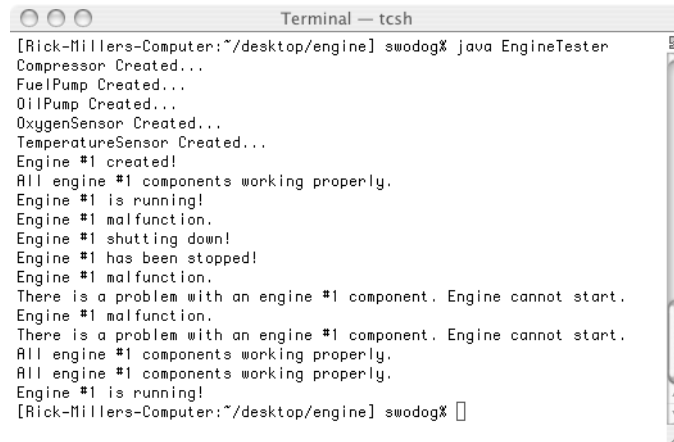
```

1      public class EngineTester {
2          public static void main(String[] args){
3              Engine e1 = new Engine(1);
4              e1.startEngine();
5              e1.setCompressorStatus(PartStatus.NOT_WORKING);
6              e1.startEngine();
7              e1.startEngine();
8              e1.setCompressorStatus(PartStatus.WORKING);
9              e1.startEngine();
10         }
11     }

```

Referring to example 10.8 — line 3 starts the program by creating an `Engine` object. As noted above this line kicks off the sequence of events illustrated in figure 10-13. All the `Engine`'s part objects are initially created with `WORKING` part status. The `startEngine()` method call on line 4 initiates a `checkEngineStatus()` method call. (*Refer to the `Engine` class code listing in the next section.*) If all goes well the engine reports that it is running.

On line 5 a fault is introduced to the engine's compressor component. This causes the engine to shut down. Attempts on lines 6 and 7 to start the engine fail due to the failed compressor. When the compressor fault is removed the engine starts fine. This sequence of events can be traced through the source code and by carefully reading the program's output shown in figure 10-14.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/engine] swodog% java EngineTester
Compressor Created...
FuelPump Created...
OilPump Created...
OxygenSensor Created...
TemperatureSensor Created...
Engine #1 created!
All engine #1 components working properly.
Engine #1 is running!
Engine #1 malfunction.
Engine #1 shutting down!
Engine #1 has been stopped!
Engine #1 malfunction.
There is a problem with an engine #1 component. Engine cannot start.
Engine #1 malfunction.
There is a problem with an engine #1 component. Engine cannot start.
All engine #1 components working properly.
All engine #1 components working properly.
Engine #1 is running!
[Rick-Millers-Computer:~/desktop/engine] swodog%

```

Figure 10-14: Result of Running Example 10.8

Quick Review

The `Engine` class is a composite aggregate. Its behavior is dictated by the behavior of its contained part class objects. Its part classes include `Compressor`, `FuelPump`, `OilPump`, `OxygenSensor`, and `TemperatureSensor`.

All the classes in the aircraft simulation use the functionality of the `PartStatus` class. The `PartStatus` class employs the typesafe enumeration design pattern. Use the typesafe enumeration pattern when you want the compiler to perform state-value assignment validation.

COMPLETE AIRCRAFT ENGINE SIMULATION CODE LISTING

10.9 Engine.java

```

1     public class Engine {
2         private Compressor its_compressor = null;
3         private FuelPump its_fuelpump = null;
4         private OilPump its_oilpump = null;
5         private OxygenSensor its_oxygensensor = null;
6         private TemperatureSensor its_temperaturesensor = null;
7         private int its_engine_number = 0;
8         private boolean is_running = false;
9         private PartStatus its_status = null;
10
11        public Engine(int engine_number) {
12            its_engine_number = engine_number;
13            its_compressor = new Compressor(PartStatus.WORKING, its_engine_number);
14            its_fuelpump = new FuelPump(PartStatus.WORKING, its_engine_number);
15            its_oilpump = new OilPump(PartStatus.WORKING, its_engine_number);
16            its_oxygensensor = new OxygenSensor(PartStatus.WORKING, its_engine_number);
17            its_temperaturesensor = new TemperatureSensor(PartStatus.WORKING, its_engine_number);
18            its_status = PartStatus.WORKING;
19            System.out.println("Engine #" + its_engine_number + " created!");
20        }
21
22        public void setCompressorStatus(PartStatus status){
23            its_compressor.setPartStatus(status);
24            checkEngineStatus();
25        }
26
27        public void setFuelPumpStatus(PartStatus status){
28            its_fuelpump.setPartStatus(status);
29            checkEngineStatus();
30        }
31    }
32
33    public void setOilPumpStatus(PartStatus status){
34        its_oilpump.setPartStatus(status);
35        checkEngineStatus();
36    }
37
38    public void setOxygenSensorStatus(PartStatus status){
39        its_oxygensensor.setPartStatus(status);
40        checkEngineStatus();
41    }
42
43    public void setTemperatureSensor(PartStatus status){
44        its_temperaturesensor.setPartStatus(status);
45        checkEngineStatus();
46    }
47
48    public boolean checkEngineStatus(){
49        if(its_compressor.isWorking() && its_fuelpump.isWorking() &&
50            its_oilpump.isWorking() && its_oxygensensor.isWorking() &&
51            its_temperaturesensor.isWorking()){
52            its_status = PartStatus.WORKING;
53            System.out.println("All engine #" + its_engine_number + " components working properly.");
54        } else {
55            its_status = PartStatus.NOT_WORKING;
56            System.out.println("Engine #" + its_engine_number + " malfunction.");
57            if(is_running){
58                System.out.println("Engine #" + its_engine_number + " shutting down!");
59                stopEngine();
60            }
61        }
62
63        return its_status.isWorking();
64    }
65
66    public void startEngine(){
67        if(!is_running){
68            if(checkEngineStatus()){
69                is_running = true;
70                System.out.println("Engine #" + its_engine_number + " is running!");
71            } else {
72                System.out.println("There is a problem with an engine #" + its_engine_number
73                    + " component. Engine cannot start.");
74            }

```



```

75     } else {
76         System.out.println("Engine #" + its_engine_number + " is already running!");
77     }
78 }
79
80 public void stopEngine(){
81     is_running = false;
82     System.out.println("Engine #" + its_engine_number + " has been stopped!");
83 }
84
85 }

```

10.10 Compressor.java

```

1 public class Compressor {
2     private int registered_engine_number = 0;
3     private PartStatus part_status;
4
5     public Compressor(PartStatus status, int engine_number){
6         registered_engine_number = engine_number;
7         part_status = status;
8         System.out.println("Compressor Created...");
9     }
10
11     public boolean isWorking(){ return part_status.isWorking(); }
12
13     public void setPartStatus(PartStatus status){
14         part_status = status;
15     }
16
17     public void setRegisteredEngineNumber(int engine_number){
18         registered_engine_number = engine_number;
19     }
20
21     public int getRegisteredEngineNumber(){ return registered_engine_number; }
22 }

```

10.11 FuelPump.java

```

1 public class FuelPump {
2     private int registered_engine_number = 0;
3     private PartStatus part_status;
4
5     public FuelPump(PartStatus status, int engine_number){
6         registered_engine_number = engine_number;
7         part_status = status;
8         System.out.println("FuelPump Created...");
9     }
10
11     public boolean isWorking(){ return part_status.isWorking(); }
12
13     public void setPartStatus(PartStatus status){
14         part_status = status;
15     }
16
17     public void setRegisteredEngineNumber(int engine_number){
18         registered_engine_number = engine_number;
19     }
20
21     public int getRegisteredEngineNumber(){ return registered_engine_number; }
22 }

```

10.12 OilPump.java

```

1 public class OilPump {
2     private int registered_engine_number = 0;
3     private PartStatus part_status;
4
5     public OilPump(PartStatus status, int engine_number){
6         registered_engine_number = engine_number;
7         part_status = status;
8         System.out.println("OilPump Created...");
9     }
10
11     public boolean isWorking(){ return part_status.isWorking(); }
12
13     public void setPartStatus(PartStatus status){
14         part_status = status;
15     }
16
17     public void setRegisteredEngineNumber(int engine_number){

```

```

18     registered_engine_number = engine_number;
19     }
20
21     public int getRegisteredEngineNumber(){ return registered_engine_number; }
22 }

```

10.13 OxygenSensor.java

```

1     public class OxygenSensor {
2         private int registered_engine_number = 0;
3         private PartStatus part_status;
4
5         public OxygenSensor(PartStatus status, int engine_number){
6             registered_engine_number = engine_number;
7             part_status = status;
8             System.out.println("OxygenSensor Created...");
9         }
10
11        public boolean isWorking(){ return part_status.isWorking(); }
12
13        public void setPartStatus(PartStatus status){
14            part_status = status;
15        }
16
17        public void setRegisteredEngineNumber(int engine_number){
18            registered_engine_number = engine_number;
19        }
20
21        public int getRegisteredEngineNumber(){ return registered_engine_number; }
22    }

```

10.14 TemperatureSensor.java

```

1     public class TemperatureSensor {
2         private int registered_engine_number = 0;
3         private PartStatus part_status;
4
5         public TemperatureSensor(PartStatus status, int engine_number){
6             registered_engine_number = engine_number;
7             part_status = status;
8             System.out.println("TemperatureSensor Created...");
9         }
10
11        public boolean isWorking(){ return part_status.isWorking(); }
12
13        public void setPartStatus(PartStatus status){
14            part_status = status;
15        }
16
17        public void setRegisteredEngineNumber(int engine_number){
18            registered_engine_number = engine_number;
19        }
20
21        public int getRegisteredEngineNumber(){ return registered_engine_number; }
22    }

```

10.15 PartStatus.java

```

1     public class PartStatus {
2         private boolean its_status = false;
3
4         public static final PartStatus WORKING = new PartStatus(true);
5         public static final PartStatus NOT_WORKING = new PartStatus(false);
6
7         private PartStatus(boolean status){
8             its_status = status;
9         }
10
11        public boolean isWorking(){ return its_status; }
12
13    }

```

10.16 EngineTester.java

```

1     public class EngineTester {
2         public static void main(String[] args){
3             Engine e1 = new Engine(1);
4             e1.startEngine();
5             e1.setCompressorStatus(PartStatus.NOT_WORKING);
6             e1.startEngine();
7             e1.startEngine();
8             e1.setCompressorStatus(PartStatus.WORKING);
9             e1.startEngine();
10        }
11    }

```

SUMMARY

There are two types of complexity: conceptual and physical. Conceptual complexity is managed by using object-oriented concepts and expressing your object-oriented designs in the Unified Modeling Language (UML). Physical complexity can be managed by your IDE or with an automated build tool such as Ant. You can also manage physical complexity on a small scale by organizing your source files into project directories and compiling multiple files simultaneously using the javac compiler tool.

A dependency is a relationship between two classes in which a change to the depended-upon class will affect the dependent class. An association is a peer-to-peer structural link between two classes.

An aggregation is a special type of association between two objects that results in a whole/part relationship. There are two types of aggregation: simple and composite. The type of aggregation is determined by the extent to which the whole object controls the lifetime of its part objects. If the whole object simply uses the services of a part object but does not control its lifetime then it's a simple aggregation. On the other hand, if the whole object creates and controls the lifetime of its part objects then it is a composite aggregate.

The UML class diagram can be used to show aggregation associations between classes. A hollow diamond is used to denote simple aggregation and expresses a “uses” or “uses a” relationship between the whole and part classes. A solid diamond is used to show composite aggregation and expresses a “has” or “has a” relationship between whole and part classes.

Sequence diagrams are a type of UML diagram used to graphically illustrate a sequence of system events. Sequence event participants can be internal system objects or external actors. Sequence diagrams do a good job of illustrating the complex message passing between objects that participate in a compositional design.

The Engine class is a composite aggregate. Its behavior is dictated by the behavior of its contained part class objects. Its part classes include Compressor, FuelPump, OilPump, OxygenSensor, and TemperatureSensor.

All the classes in the aircraft simulation use the functionality of the PartStatus class. The PartStatus class employs the typesafe enumeration design pattern. Use the typesafe enumeration design pattern when you want the compiler to perform state-value assignment validation.

Skill-Building Exercises

1. **Study The UML:** Obtain a reference on the Unified Modeling Language (UML) and conduct further research on how to model complex associations and aggregations using class diagrams. Also study how to express complex event sequences using sequence diagrams.
2. **Obtain a UML Modeling Tool:** If you haven't already done so procure a UML modeling tool to help you draw class and sequence diagrams.
3. **Research the Typesafe Enumeration Pattern:** Conduct a web search to learn more about the use of the typesafe enumeration pattern.
4. **Discover Dependency Relationships:** Study the Java Platform API. Write down at least five instances where one class depends on the services of another class. Describe the nature of the dependency and explain the possible effects changing the depended-upon class might have on the dependent class.
5. **Programming Exercise — Demonstrate Simple Aggregation:** Write a program that implements the simple aggregation shown in figure 10-15.

Hints: Implement the `printMessage()` method in `ServerClass` to print a short message of your choosing to the console. You'll need to write a test driver program to test your code. Study examples 10.1 through 10.3 for clues on how to implement this exercise.

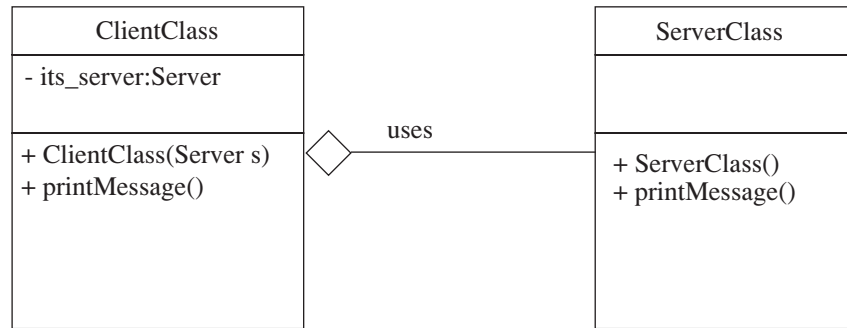


Figure 10-15: Simple Aggregation Class Diagram

6. Programming Exercise — Demonstrate Composite Aggregation: Write a program that implements the composite aggregation shown in figure 10-16.

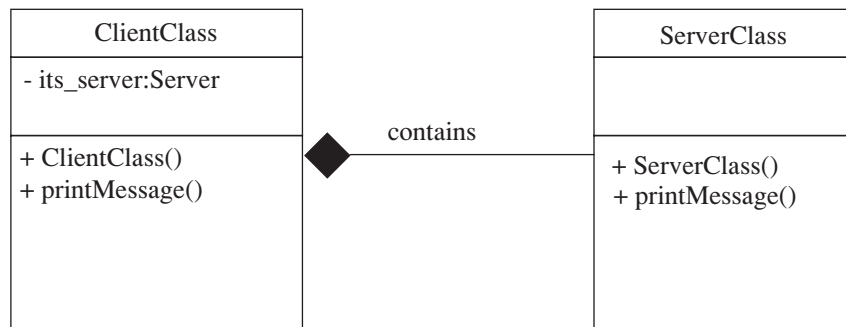


Figure 10-16: Composite Aggregation Class Diagram

Hints: Implement the `printMessage()` method in `ServerClass` to print a short message of your choosing to the console. You'll need to write a test driver program to test your code. Study examples 10.4 through 10.6 for clues on how to implement this exercise.

7. Compile Multiple Source Files: Find the aircraft project simulator project source code presented in this chapter on the supplemental CD-ROM. Use the `javac` compiler tool to compile all the source files in the project directory using the `*` wildcard character.

SUGGESTED PROJECTS

- 1. Create Sequence Diagrams For Aircraft Engine Simulator:** Using your UML modeling tool create a set of sequence diagrams to model the `startEngine()`, `stopEngine()`, and `checkEngineStatus()` method events.
- 2. Programming Project — Aircraft Simulator:** Using the code supplied in the aircraft engine simulation project write a program that simulates aircraft with various numbers of engines. Give your aircraft the ability to start, stop, and check the status of each engine. Implement the capability to inject engine part faults. Utilize a simple text-based menu to control your aircraft's engines. Draw a UML class diagram of your intended design.
- 3. Programming Project — Automobile Simulator:** Write a program that implements a simple automobile simulation. Perform the analysis and determine what parts your simulated car needs. Modify the aircraft engine simulator

code to use in this project or adopt it as-is. Control your car with a simple text-based menu. Draw a UML class diagram of your intended design.

4. **Programming Project — Gas Pump Simulation:** Write a program that implements a simple gas pump system. Perform the analysis to determine what components your gas pump system requires. Control your gas pump system with a simple text-based menu. Draw a UML class diagram of your intended design.
5. **Programming Project — Hubble Telescope:** Write a program that controls the Hubble Space Telescope. Perform the analysis to determine what components the telescope control system will need. Control the space telescope with a simple text-based menu. Draw a UML class diagram of your intended design.
6. **Programming Project — Mars Rover:** Write a program that controls the Mars Rover. Perform the analysis to determine what components your rover requires. Control the rover with a simple text-based menu. Draw a UML class diagram of your intended design.

SELF-TEST QUESTIONS

1. What is a dependency relationship between two objects?
2. What is an association relationship between two objects?
3. What is an aggregation?
4. List the two types of aggregation.
5. Explain the difference between the two types of aggregation.
6. How do you express simple aggregation using a UML class diagram?
7. How do you express composite aggregation using a UML class diagram?
8. Which type of aggregation denotes a *uses* or *uses a* relationship between two objects?
9. Which type of aggregation denotes a *has* or *has a* relationship between two objects?
10. What is the purpose of a UML sequence diagram?
11. Explain the utility of the typesafe enumeration pattern.
12. A class built from other class types is referred to as a/an _____.
13. In this type of aggregation part objects belong solely to the whole or containing class. Name the type of aggregation.
14. In this type of aggregation part object lifetimes are not controlled by the whole or containing class. Name the type of aggregation.
15. In a UML class diagram the aggregation diamond is drawn closest to which class, the whole or the part?

REFERENCES

Grady Booch, et. al. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1998. ISBN: 0-201-57168-4

Rick Miller. *C++ For Artists: The Art, Philosophy, And Science of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA, 2003. ISBN: 1-932504-02-8

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Second Edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994. ISBN: 0-8053-5340-2

Erik Hatcher, et. al. *Java Development with Ant*. Manning Publications Co. Greenwich, CT. ISBN: 1-930110-58-8

NOTES

CHAPTER 11



Washington Canoe Club

EXTENDING CLASS BEHAVIOR THROUGH INHERITANCE

LEARNING OBJECTIVES

- *STATE THE PURPOSE OF INHERITANCE*
- *STATE THE PURPOSE OF A BASE CLASS*
- *STATE THE PURPOSE OF A DERIVED CLASS*
- *STATE THE PURPOSE OF AN ABSTRACT METHOD*
- *STATE THE PURPOSE OF AN ABSTRACT BASE CLASS*
- *STATE THE PURPOSE OF AN INTERFACE*
- *DEMONSTRATE YOUR ABILITY TO WRITE POLYMORPHIC JAVA CODE*
- *STATE THE PURPOSE OF A DEFAULT CONSTRUCTOR*
- *DESCRIBE THE BEHAVIOR OF CONSTRUCTOR CHAINING*
- *DEMONSTRATE YOUR ABILITY TO CREATE A UML DIAGRAM THAT ILLUSTRATES INHERITANCE RELATIONSHIPS BETWEEN JAVA CLASSES*
- *STATE THE PURPOSE OF THE FINAL KEYWORD*
- *DEMONSTRATE YOUR ABILITY TO UTILIZE ABSTRACT METHODS, ABSTRACT CLASSES, AND INTERFACES IN YOUR JAVA PROGRAMS*
- *DEMONSTRATE YOUR ABILITY TO UTILIZE FINAL METHODS AND CLASSES IN YOUR JAVA PROGRAMS.*

INTRODUCTION

Inheritance is a powerful object-oriented programming feature provided by the Java programming language. The behavior provided by or specified by one class (*base class*) can be adopted or extended by another class (*derived or subclass*). Up to this point you have been using inheritance in every Java program you have written although mostly this has been done for you by the Java compiler and platform behind the scenes. For example, every user-defined class you create automatically inherits from the `java.lang.Object` class. (See chapter 8, figure 8-3 or chapter 9, figure 9-2)

In this chapter you will learn how to create new derived classes from existing classes and interfaces. There are three ways of doing this: 1) by *extending* the functionality of an existing class, 2) by *implementing* one or more interfaces, or 3) you can combine these two methods to create derived classes that both extend the functionality of a base class and implement the operations declared in one or more interfaces.

Along the way I will show you how to create and use abstract methods to create *abstract classes*. You will learn how to create and utilize *interfaces* in your program designs as well as how to employ the *final* keyword to inhibit the inheritance mechanism. You will also learn how to use a UML class diagram to show inheritance hierarchies.

By the time you complete this chapter you will fully understand how to create an object-oriented Java program that exhibits *dynamic polymorphic behavior*. Most importantly, however, you will understand why dynamic polymorphic behavior is the desired end-state of an object-oriented program.

This chapter also builds on the material presented in chapter 10 - Compositional Design. The primary chapter example demonstrates the design possibilities you can achieve when you combine inheritance with compositional design.

THREE PURPOSES OF INHERITANCE

Inheritance serves three essential purposes. The first purpose of inheritance is to serve as an object-oriented design mechanism that enables you to think and reason about the structure of your programs in terms of *generalized* and *specialized* class behavior. A base class implements, or specifies, generalized behavior common to all of its subclasses. Subclasses derived from this base class capitalize on the behavior it provides. Additionally, subclasses may specify, or implement, specialized behavior if required in the context of the design.

When you think in terms of inheritance you think in terms of class hierarchies where the base classes that implement generalized behavior appear at or near the top of the hierarchy and the derived classes that implement specialized behavior appear toward the bottom. Figure 11-1 gives a classic example of an inheritance hierarchy showing generalized/specialized behavior.

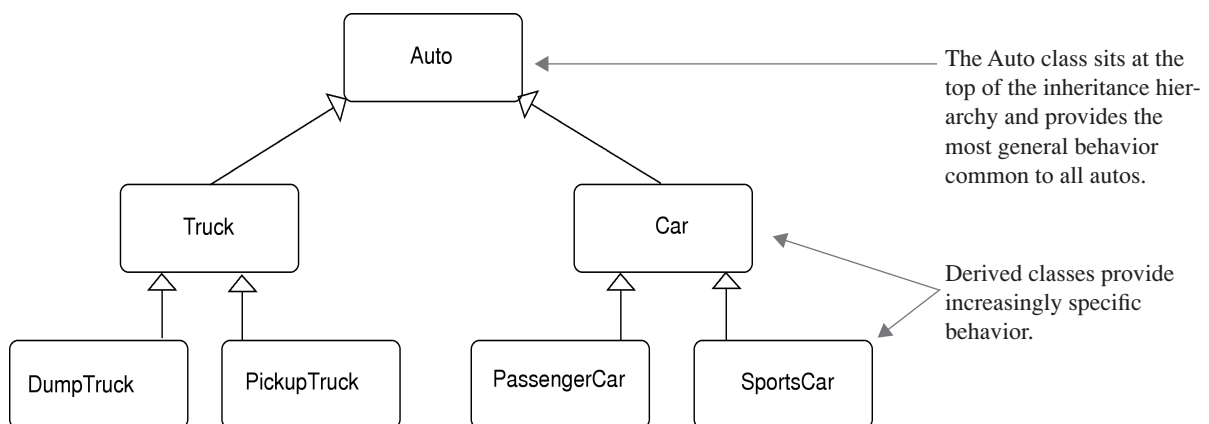


Figure 11-1: Inheritance Hierarchy Illustrating Generalized & Specialized Behavior

Referring to figure 11-1 — the Auto class sits at the top of the inheritance hierarchy and provides generalized behavior for all of its derived classes. The Truck and Car classes derive from Auto. They may provide specialized behavior found only in trucks and cars. The DumpTruck and PickupTruck classes derive from Truck, which means that Truck is also serving as a base class. DumpTruck and PickupTruck inherit Truck's generalized behavior and implement the specialized behavior required of these classes. The same holds true for the PassengerCar and SportsCar classes. Their *direct base class* is Car, whose direct base class is Auto. For more real-world examples of inheritance hierarchies simply consult the Java API documentation or refer to chapter 5.

The second purpose of inheritance is to provide a way to gain a measure of code reuse within your programs. If you can implement generalized behavior in a base class that's common to all of its subclasses then you don't have to re-write the code in each subclass. If, in one of your programming projects, you create an inheritance hierarchy and find you are repeating a lot of the same code in each of the subclasses, then it's time for you to refactor your design and migrate the common code into a base class higher up in your inheritance hierarchy.

The third purpose of inheritance is to enable you to incrementally develop code. It is rare for a programmer, or more often, a team of programmers, to sit down and in one heroic effort completely code the entire inheritance hierarchy for a particular application. It's more likely the case that the inheritance hierarchy, and its accompanying classes, grows over time. (*Take the Java API as a prime example.*)

IMPLEMENTING THE “IS A” RELATIONSHIP

A class that belongs to an inheritance hierarchy participates in what is called an “*is a*” relationship. Referring again to figure 11-1, a Truck is an Auto and a Car is an Auto. Likewise, a DumpTruck is a Truck, and since a Truck is an Auto a DumpTruck is an Auto as well. In other words, class hierarchies are transitive in nature when navigating from specialized classes to more generalized classes. They are not transitive in the opposite direction however. For instance, an Auto is not a Truck or a Car, etc.

A thorough understanding of the “*is a*” relationships that exist within an inheritance hierarchy will pay huge dividends when you want to substitute a derived class object in code that specifies one of its base classes. (*This is a critical skill in Java programming because it's done extensively in the API.*)

THE RELATIONSHIP BETWEEN THE TERMS TYPE, INTERFACE, AND CLASS

Before moving on it will help you to understand the relationship between the terms *type*, *class* and *interface*. Java is a strongly-typed programming language. This means that when you write a Java program and wish to call a method on a particular object, Java must know, in advance, the type of object to which you refer. In this context the term *type* refers to *that set of operations or methods a particular object supports*. Every object you use in your Java programs has an associated type. If, by mistake, you try and call a non-supported method on an object, you will be alerted to your mistake by a compiler error when you try and compile your program.

MEANING OF THE TERM INTERFACE

An interface is a Java construct that introduces a new data type and its set of authorized operations in the form of method declarations. A method declaration specifies the method signature but omits the method body. No body — no behavior. Interfaces are discussed in more detail later in the chapter.

MEANING OF THE TERM CLASS

A class is a Java construct that introduces and defines a new data type. Like the interface, the class specifies a set of legal operations that can be called on an object of its type. However, the class can go one step further and provide definitions (*i.e., behavior*) for some or all of its methods. When a class provides definitions for all of its methods it is a *concrete class*, meaning that objects of that class type can be created with the new operator. (*i.e., they can be instantiated*) If a class definition omits the body, and therefore the behavior, of one or more of its methods, then that class must be declared to be an *abstract class*. Abstract class objects cannot be created with the new operator. I will discuss abstract classes in greater detail later in the chapter.

Quick Review

Inheritance serves three essential purposes: 1) it is an object-oriented design mechanism that enables you to think and reason about your program structure in terms of generalized and specialized class behavior, 2) it provides you with a measure of code reuse within your program by locating common class behavior in base classes, and 3) it provides a means to incrementally develop your programs over time.

Classes that belong to an inheritance hierarchy participate in an “is a” relationship between themselves and their chain of base classes. This “is a” relationship is transitive in the direction of specialized to generalized classes but not vice versa.

The Java class and interface constructs are each used to create new, user-defined data types. The interface construct is used to specify a set of authorized type methods and omits method behavior; the class construct is used to specify a set of authorized type methods and their behavior. A class construct, like an interface, can omit the bodies of one or more of its methods, however, such methods must be declared to be abstract. A class that declares one or more of its methods to be abstract must itself be declared to be an abstract class. Abstract class objects cannot be created with the new operator.

EXPRESSING GENERALIZATION & SPECIALIZATION IN THE UML

Generalization & specialization relationships can be expressed in a UML class diagram by drawing a solid line with a hollow-tipped arrow from the derived class to the base class as figure 11-2 illustrates.

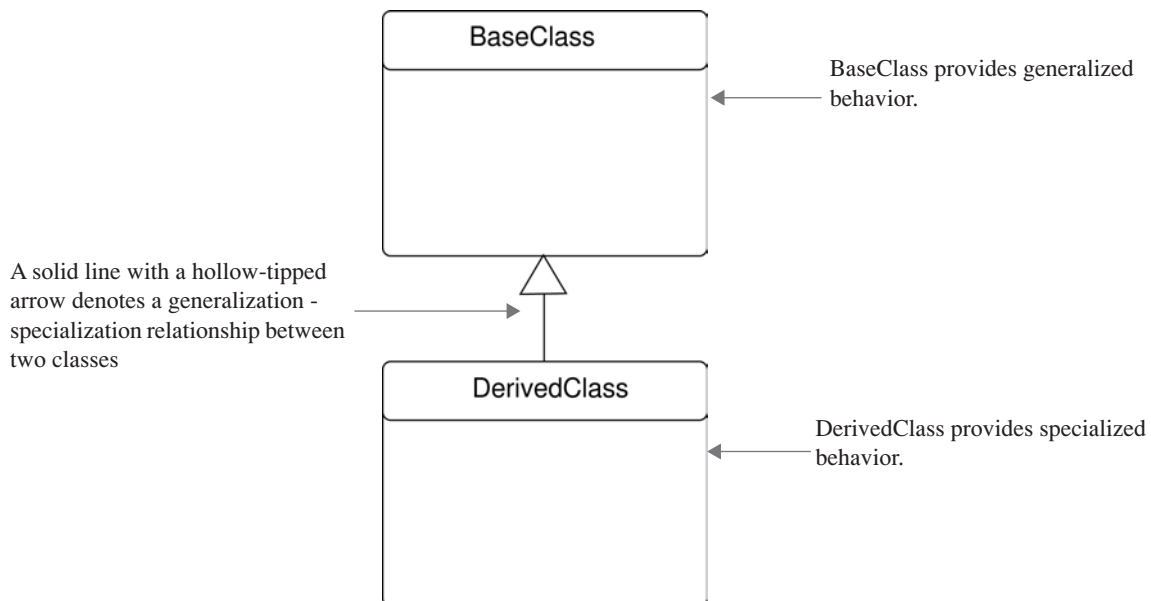


Figure 11-2: UML Class Diagram Showing DerivedClass Inheriting From BaseClass

Referring to figure 11-2 — the BaseClass class acts as the direct base class to DerivedClass. Behavior provided by BaseClass is inherited by DerivedClass. Let’s take a look at an example program that implements these two classes.

A SIMPLE INHERITANCE EXAMPLE

The simple inheritance example program presented in this section expands on the UML diagram shown in figure 11-2. The behavior implemented by BaseClass is kept intentionally simple so you can concentrate on the topic of inheritance. You'll be introduced to more complex programs soon enough.

THE UML DIAGRAM

A more complete UML diagram showing the fields and methods of BaseClass and DerivedClass is presented in figure 11-3

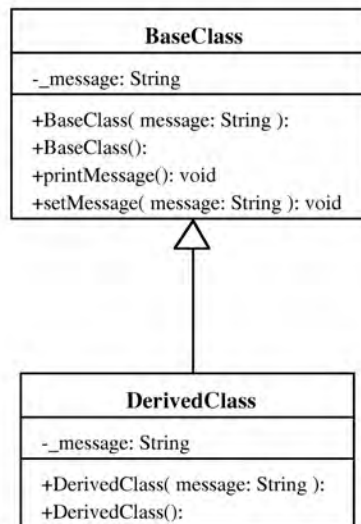


Figure 11-3: UML Diagram of BaseClass & DerivedClass Showing Fields and Methods

Referring to figure 11-3 — BaseClass contains one private field named `_message` which is of type `String`. BaseClass has four public methods: two constructors, `printMessage()`, and `setMessage()`. One of the constructors is a default constructor. A default constructor is simply a constructor that takes no arguments. The second constructor has one parameter of type `String` named `message`. Based on this information, objects of type `BaseClass` can be created in two ways. Once an object of type `BaseClass` is created the `printMessage()` or the `setMessage()` methods can be called on that object.

DerivedClass has its own private `_message` field and two constructors that are similar to the constructors found in BaseClass. Let's now take a look at the source code for each class.

BASECLASS SOURCE CODE

```

1      public class BaseClass {
2          private String _message = null;
3
4          public BaseClass(String message) {
5              _message = "Message from BaseClass: " + message;
6          }
7
8          public BaseClass() {
9              this("BaseClass message!");
10         }
11
12         public void printMessage() {
  
```

11.1 BaseClass.java

```

13         System.out.println(_message);
14     }
15
16     public void setMessage(String message) {
17         _message = "Message from BaseClass: " + message;
18     }
19 }

```

Referring to example 11.1 — `BaseClass` is fairly simple. Its first constructor begins on line 4 and declares one `String` parameter named `message`. The `_message` field is set by concatenating the `message` parameter and the `String` literal “*Message from BaseClass:*”. The default constructor begins on line 8. It calls the first constructor with the `String` literal “*BaseClass message!*”. The `printMessage()` method begins on line 12. It simply prints the `_message` field to the console. The `setMessage()` method begins on line 16. Its job is to change the value of the `_message` field.

Since all methods have a body, and are therefore defined, the `BaseClass` is also a concrete class. This means that objects of type `BaseClass` can be created with the new operator.

DERIVEDCLASS SOURCE CODE

11.2 *DerivedClass.java*

```

1     public class DerivedClass extends BaseClass {
2         private String _message = null;
3
4         public DerivedClass(String message) {
5             super(message);
6             _message = message;
7         }
8
9         public DerivedClass() {
10            this("DerivedClass message!");
11        }
12    }

```

Referring to example 11.2 — `DerivedClass` inherits the functionality of `BaseClass` by extending `BaseClass` using the *extends* keyword on line 1. `DerivedClass` itself provides two constructors and a private field named `_message`. The first constructor begins on line 4. It declares a `String` parameter named `message`. The first thing this constructor does, however, on line 5, is call the `String` parameter version of the `BaseClass` constructor using the `super()` method with the `message` parameter as an argument. If the `super()` method is called in a derived class constructor it must be the first line of code in the constructor body as is done here. The next thing the `DerivedClass` constructor does is set the value of its `_message` field to the value of the `message` parameter. (*Remember, these are references to String objects.*)

`DerivedClass`’s default constructor begins on line 9. It calls its version of the `String` parameter constructor using the `String` literal “*DerivedClass message!*” as an argument, which, as you learned above, then calls the `BaseClass` constructor via the `super()` method.

Let’s now take a look at how these two classes can be used in a program.

DRIVERAPPLICATION PROGRAM

11.3 *DriverApplication.java*

```

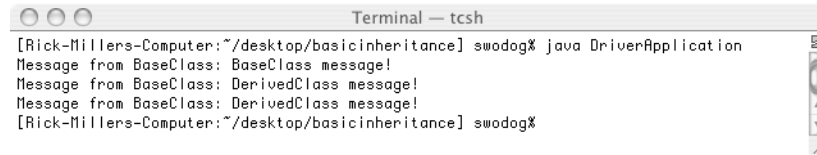
1     public class DriverApplication {
2         public static void main(String[] args) {
3             BaseClass bcr1 = new BaseClass();
4             BaseClass bcr2 = new DerivedClass();
5             DerivedClass dcr1 = new DerivedClass();
6
7             bcr1.printMessage();
8             bcr2.printMessage();
9             dcr1.printMessage();
10        }
11    }

```

The `DriverApplication` class is used to test the functionality of `BaseClass` and `DerivedClass`. The important thing to note in this example is what type of object is being declared and created on lines 3 through 5. Starting on line 3 a `BaseClass` reference named `bcr1` is declared and initialized to point to a `BaseClass` object. On line 4 another `BaseClass` reference named `bcr2` is declared and initialized to point to a `DerivedClass` object. On line 5 a `DerivedClass` reference named `dcr1` is declared and initialized to point to a `DerivedClass` object. Note that a reference to a base class

object can point to a derived class object. Also note that for this example only the default constructors are being used to create each object. This will result in the default message text being assigned to each object's `_message` field.

Continuing with example 11.3 — on lines 7 through 9 the `printMessage()` method is called on each reference. It's time now to compile and run the code. Figure 11-4 gives the results of running example 11.3.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/basicinheritance] swodog% java DriverApplication
Message from BaseClass: BaseClass message!
Message from BaseClass: DerivedClass message!
Message from BaseClass: DerivedClass message!
[Rick-Millers-Computer:~/desktop/basicinheritance] swodog%
```

Figure 11-4: Results of Running Example 11.3

As you will notice from studying figure 11-4 there are three lines of program output that correspond to the three `printMessage()` method calls on each reference `bcr1`, `bcr2`, and `dcr1`. Creating a `BaseClass` reference and initializing it to a `BaseClass` object results in the `BaseClass` version (*the only version at this point*) of the `printMessage()` method being called which prints the default `BaseClass` text message.

Creating a `BaseClass` reference and initializing it to point to a `DerivedClass` object has slightly different behavior, namely, the value of the resulting text printed to the console reflects the fact that a `DerivedClass` object was created, which resulted in the `BaseClass` `_message` field being set to the `DerivedClass` default value. Note that `DerivedClass` does not have a `printMessage()` method therefore it is the `BaseClass` version of `printMessage()` that is called. This behavior is inherited by `DerivedClass`.

Finally, creating a `DerivedClass` reference and initializing it to point to a `DerivedClass` object appears to have the same effect as the previous `BaseClass` reference -> `DerivedClass` object combination. This is the case in this simple example because `DerivedClass` simply inherits `BaseClass`'s default behavior and, except for its own constructors, leaves it unchanged.

Quick Review

A base class implements default behavior in the form of methods that can be inherited by derived classes. There are three reference -> object combinations: 1) if the base class is a concrete class, meaning it is not abstract, then a base class reference can point to a base class object, 2) a base class reference can point to a derived class object, and 3) a derived class reference can point to a derived class object.

ANOTHER INHERITANCE EXAMPLE: PERSON - STUDENT

Let's now take a look at a more real-world example of inheritance. This example will utilize the `Person` class presented in chapter 9 as a base class. The derived class will be called `Student`. Let's take a look at the UML diagram for this inheritance hierarchy.

THE PERSON - STUDENT UML CLASS DIAGRAM

Figure 11-5 gives the UML class diagram for the `Student` class inheritance hierarchy. Notice the behavior provided by the `Person` class in the form of its public interface methods. The `Student` class extends the functionality of `Person` and provides a small bit of specialized functionality of its own in the form of the `getStudentInfo()` method.

Since the `Student` class participates in an is-a relationship with class `Person`, a `Student` object can be used where a `Person` object is called for in your source code. However, now you must be keenly aware of the specialized behavior provided by the `Student` class as you will soon see when you examine and run the driver application program for this example.

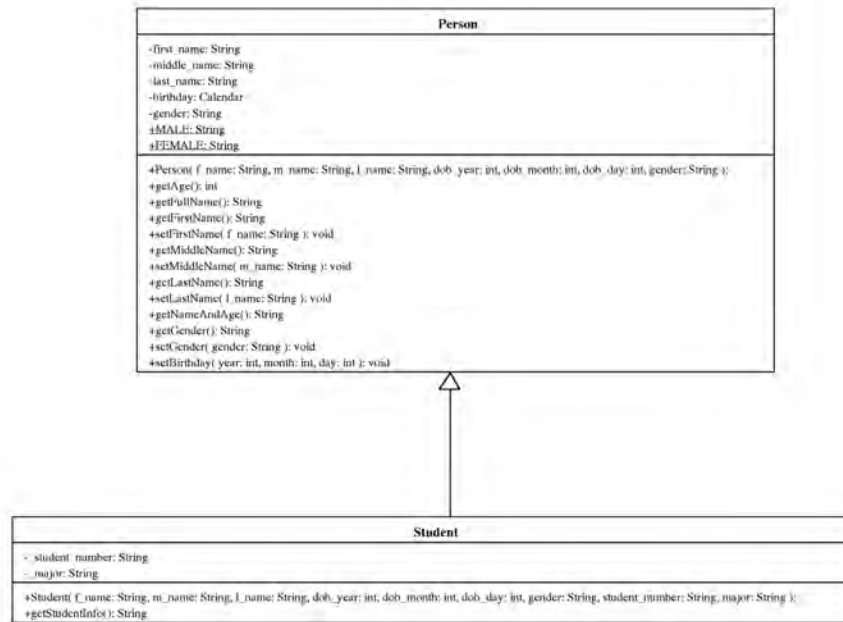


Figure 11-5: UML Diagram Showing Student Class Inheritance Hierarchy

PERSON - STUDENT SOURCE CODE

```

1      import java.util.*;
2
3      public class Person {
4          private String first_name = null;
5          private String middle_name = null;
6          private String last_name = null;
7          private Calendar birthday = null;
8          private String gender = null;
9
10         public static final String MALE = "Male";
11         public static final String FEMALE = "Female";
12
13         public Person(String f_name, String m_name, String l_name, int dob_year,
14             int dob_month, int dob_day, String gender){
15             first_name = f_name;
16             middle_name = m_name;
17             last_name = l_name;
18             this.gender = gender;
19
20             birthday = Calendar.getInstance();
21             birthday.set(dob_year, dob_month, dob_day);
22         }
23
24         public int getAge(){
25             Calendar today = Calendar.getInstance();
26             int now = today.get(Calendar.YEAR);
27             int then = birthday.get(Calendar.YEAR);
28             return (now - then);
29         }
30
31         public String getFullName(){ return (first_name + " " + middle_name + " " + last_name); }
32
33         public String getFirstName(){ return first_name; }
34         public void setFirstName(String f_name) { first_name = f_name; }
35
36         public String getMiddleName(){ return middle_name; }
37         public void setMiddleName(String m_name){ middle_name = m_name; }
38
39         public String getLastName(){ return last_name; }
  
```

11.4 Person.java

```

40     public void setLastName(String l_name){ last_name = l_name; }
41
42     public String getNameAndAge(){ return (getFullName() + " " + getAge()); }
43
44     public String getGender(){ return gender; }
45     public void setGender(String gender){ this.gender = gender; }
46
47     public void setBirthday(int year, int month, int day){ birthday.set(year, month, day); }
48
49 } //end Person class

```

The Person class code is unchanged from chapter 9.

11.5 Student.java

```

1     public class Student extends Person {
2         private String _student_number = null;
3         private String _major = null;
4
5         public Student(String f_name, String m_name, String l_name, int dob_year,
6             int dob_month, int dob_day, String gender, String student_number, String major){
7             super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender);
8             _student_number = student_number;
9             _major = major;
10        }
11
12        public String getStudentInfo(){
13            return getNameAndAge() + " " + _student_number + " " + _major;
14        }
15    } // end Student class

```

The Student class extends Person and implements specialized behavior in the form of the `getStudentInfo()` method. The Student class has one constructor that takes several String and integer parameters. With the exception of the last two parameters, `student_number` and `major`, the parameters are those required by the Person class and in fact they are used on line 7 as arguments to the `super()` method call. The parameters `student_number` and `major` are used to set a Student object's `_student_number` and `_major` fields respectively on lines 8 and 9.

The `getStudentInfo()` method begins on line 12. It returns a String that is a concatenation of the String returned by the Person object's `getNameAndAge()` method and the `_student_number` and `_major` fields. Take note specifically of the `getNameAndAge()` method call on line 13. You will frequently call public or protected base class methods from derived classes in this manner.

This is all the specialized functionality required of the Student class for this example. The majority of its functionality is provided by the Person class. Let's now take a look at these two classes in action. Example 11.6 gives the test driver program.

11.6 PersonStudentTestApp.java

```

1     public class PersonStudentTestApp {
2         public static void main(String[] args){
3
4             Person p1 = new Person("Steven", "Jay","Jones", 1946, 8, 30, Person.MALE);
5             Person p2 = new Student("Jeanie", "Sue", "Freeman", 1963, 10, 10, Person.FEMALE,
6                 "000002", "Business");
7             Student s1 = new Student("Richard", "Warren", "Miller", 1723, 2, 29, Person.MALE,
8                 "000003", "Math");
9
10            System.out.println(p1.getNameAndAge());
11            System.out.println(p2.getNameAndAge());
12            // System.out.println(p2.getStudentInfo()); // error - p2 is a Person reference
13            System.out.println(s1.getNameAndAge());
14            System.out.println(s1.getStudentInfo());
15        }
16    }

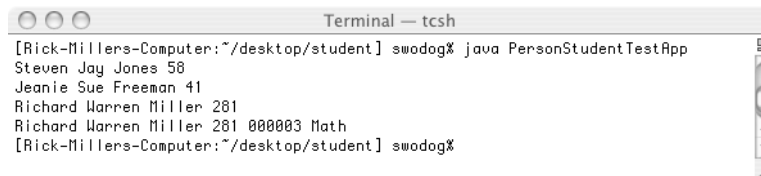
```

This program is similar in structure to example 11-3 in that it declares three references and shows you the effects of calling methods on those references. A Person reference named `p1` is declared on line 4 and initialized to point to a Person object. On line 5, another Person reference named `p2` is declared and initialized to point to a Student object. On line 7 a Student reference is declared and initialized to point to a Student object.

The method calls begin on line 10 with a call to the `getNameAndAge()` method on a Person object via the `p1` reference. This will work fine. On line 11 the `getNameAndAge()` method is called on a Student object via the `p2` reference, which, as you know, is a Person-type reference.

Line 12 is commented out. This line, if you try to compile it in its current form, will cause a compiler error because an attempt is being made to call the `getStudentInfo()` method, which is specialized behavior of the `Student` class, on a `Student` object, via a `Person`-type reference. Now, repeat the previous sentence to yourself several times until you fully understand its meaning. Good! Now, you may ask, and rightly so at this point, “But wait...why can’t you call the `getStudentInfo()` method on a `Student` object?” The answer is — you can, but `p2` is a `Person`-type reference, which means that the compiler is checking to ensure you only call methods defined by the `Person` class via that reference. Remember the “*Java is a strongly-typed language...*” spiel I delivered earlier in this chapter? I will show you how to use casting to resolve this issue after I show you how this program runs.

Continuing with example 11.6 — on line 13 the `getNameAndAge()` method is called on a `Student` object via a `Student` reference. This is perfectly fine since a `Student` is a `Person`. On line 14 the `getStudentInfo()` method is called on a `Student` object via a `Student` reference. This also performs as expected. Figure 11-6 gives the results of running example 11.6.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/student] swodog% java PersonStudentTestApp
Steven Jay Jones 58
Jeanie Sue Freeman 41
Richard Warren Miller 281
Richard Warren Miller 281 000003 Math
[Rick-Millers-Computer:~/desktop/student] swodog%
```

Figure 11-6: Results of Running Example 11.6

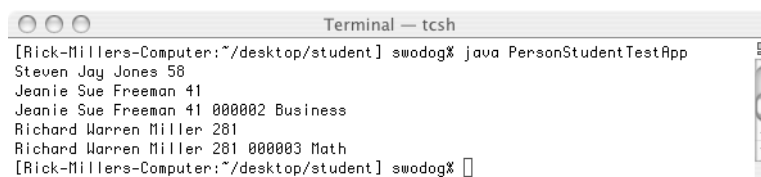
CASTING

OK, now let’s take a look at a modified version of example 11.6 that takes care of the problem encountered on line 12. Example 11.7 gives the modified version of `PersonStudentTestApp.java`.

11.7 *PersonStudentTestApp.java (Mod 1)*

```
1 public class PersonStudentTestApp {
2     public static void main(String[] args){
3
4         Person p1 = new Person("Steven", "Jay","Jones", 1946, 8, 30, Person.MALE);
5         Person p2 = new Student("Jeanie", "Sue", "Freeman", 1963, 10, 10, Person.FEMALE,
6             "000002", "Business");
7         Student s1 = new Student("Richard", "Warren", "Miller", 1723, 2, 29, Person.MALE,
8             "000003", "Math");
9
10        System.out.println(p1.getNameAndAge());
11        System.out.println(p2.getNameAndAge());
12        System.out.println(((Student)p2).getStudentInfo()); // casting resolves the issue
13        System.out.println(s1.getNameAndAge());
14        System.out.println(s1.getStudentInfo());
15    }
16 }
```

Notice on line 12 that the compiler has been instructed to treat the `p2` reference as though it were a `Student` reference. This form of explicit type coercion is called *casting*. Casting only works if the object really is of the type you are casting it to. In other words, you would be in big trouble if you tried to cast a `Person` to a `Car` since a `Person` is not a `Car`. Figure 11-7 shows the results of running this program.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/student] swodog% java PersonStudentTestApp
Steven Jay Jones 58
Jeanie Sue Freeman 41
Jeanie Sue Freeman 41 000002 Business
Richard Warren Miller 281
Richard Warren Miller 281 000003 Math
[Rick-Millers-Computer:~/desktop/student] swodog%
```

Figure 11-7: Results of Running Example 11.7

USE CASTING SPARINGLY

Casting is a helpful feature but too much casting usually means your design is not optimal from an object-oriented point of view. You will see more situations in this book where casting is required, but mostly, I will try and show you how to design programs that minimize the need to cast.

Quick Review

The Person class provided the default behavior for the Student class. The Student class inherited Person's default behavior and implemented some specialized behavior of its own. Reference variables have an associated type. Method calls to an object pointed to by a reference will succeed without casting so long as the type of the reference supports the method you are trying to call. You can force, or coerce, the compiler to treat a reference to an object of one type as if it were a reference to an object of another. This is extremely helpful in some circumstances but, as a rule, casting should be used sparingly. Also, casting only works if the object really is of the type you are casting it to.

OVERRIDING BASE CLASS METHODS

So far you have only seen examples of inheritance where the derived class fully accepted the behavior provided by its base class. This section will show you how to override base class behavior in the derived class by overriding base class methods.

To override a base class method in a derived class you will need to re-define the method with the exact signature in the derived class. The overriding derived class method must also return the same type as the overridden base class method. (*The Java compiler will complain if you try otherwise!*) Let's take a look at a simple example. Figure 11-8 gives a UML class diagram for the BaseClass and DerivedClass classes.

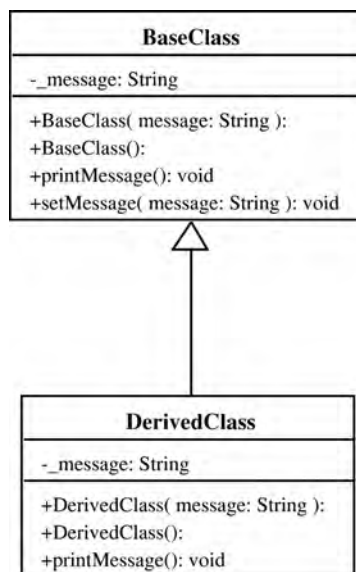


Figure 11-8: UML Class Diagram For BaseClass & DerivedClass

Referring to figure 11-8 — notice that DerivedClass now has a public method named printMessage(). BaseClass remains unchanged. Example 11.8 gives the source code for the modified version of DerivedClass.

11.8 DerivedClass.java (mod 1)

```

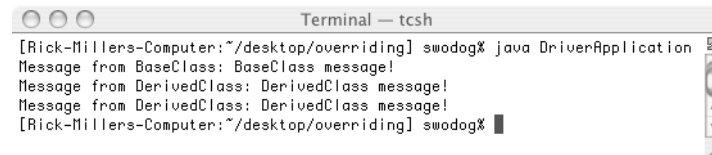
1     public class DerivedClass extends BaseClass {
2         private String _message = null;
3
4         public DerivedClass(String message) {
5             super(message);
6             _message = message;
7         }
  
```

```

8
9     public DerivedClass(){
10         this("DerivedClass message!");
11     }
12
13     public void printMessage(){
14         System.out.println("Message from DerivedClass: " + _message);
15     }
16 }

```

The only change to `DerivedClass` is the addition of the `printMessage()` method starting on line 13. `DerivedClass`'s version of `printMessage()` overrides the `BaseClass` version. How does this affect the behavior of these two classes? A good way to explore this issue is to recompile and run the `DriverApplication` given in example 11.3. Figure 11-9 shows the results of running the program using a modified version of `DerivedClass`.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/overriding] swodog% java DriverApplication
Message from BaseClass: BaseClass message!
Message from DerivedClass: DerivedClass message!
Message from DerivedClass: DerivedClass message!
[Rick-Millers-Computer:~/desktop/overriding] swodog% █

```

Figure 11-9: Results of Running Example 11.3 with Modified Version of `DerivedClass`

Referring to figure 11-9 — compare these results with those of figure 11-4. The first message is the same, which is as it should be. The `bcr1` reference points to a `BaseClass` object. The second message is different though. Why is this so? The `bcr2` reference is pointing to a `DerivedClass` object. When the `printMessage()` method is called on the `DerivedClass` object via the `BaseClass` reference the overriding `printMessage()` method provided in `DerivedClass` is called. This is an example of polymorphic behavior. A base class reference, `bcr2`, points to a derived class object. Via the base class reference you call a method provided by the base class interface but is overridden in the derived class and, voila, you have polymorphic behavior. Pretty cool, huh?

Quick Review

Derived classes can override base class behavior by providing overriding methods. An overriding method is a method in a derived class that has the same method signature as the base class method it is intending to override. Overriding methods can be called polymorphically via a base class reference that points to a derived class object.

ABSTRACT METHODS & ABSTRACT BASE CLASSES

An abstract method is one that appears in the body of a class declaration but omits the method body. A class that declares one or more abstract methods must be declared to be an abstract class. If you create an abstract method and forget to declare the class as being abstract the Java compiler will inform you of your mistake.

Now, you could simply declare a class to be abstract even though it provides implementations for all of its methods. This would prevent you from creating objects of the abstract class directly with the `new` operator. This may or may not be the intention of your application design goals. However, abstract classes of this nature are not the norm.

THE PRIMARY PURPOSE OF AN ABSTRACT BASE CLASS

The primary purpose of an abstract base class is to provide a set of one or more public interface methods whose implementations are expected to be found in some derived class further down the inheritance hierarchy. The key phrase is “*expected to be found in some derived class further down the inheritance hierarchy.*” This means that as a designer you would employ an abstract class in your application architecture when you want a base class to specify rather than implement behavior and you expect derived classes to actually implement the behavior specified by the base class interface.

OK, why would you want to do this? Why create a class that does nothing but specify a set of interface methods? Good questions! The short answer is that abstract classes will comprise the upper tier(s) of your inheritance hierarchy. The upper tier(s) of an inheritance hierarchy is where you expect to find specifications for general behavior found in derived classes which comprise the lower tier(s) of an inheritance hierarchy. The derived classes, at some point, must provide implementations for those abstract methods specified in their base classes. Designing application architectures in this fashion — abstractions at the top and concrete implementations at the bottom — enables the architecture to be *extended* to accommodate new functionality rather than modified. This design technique injects a good dose of stability into your application architecture. This and other advanced object-oriented design techniques is discussed in more detail in chapter 24.

EXPRESSING ABSTRACT BASE CLASSES IN UML

Figure 11-10 shows a UML diagram that contains an abstract base class named `AbstractClass`.

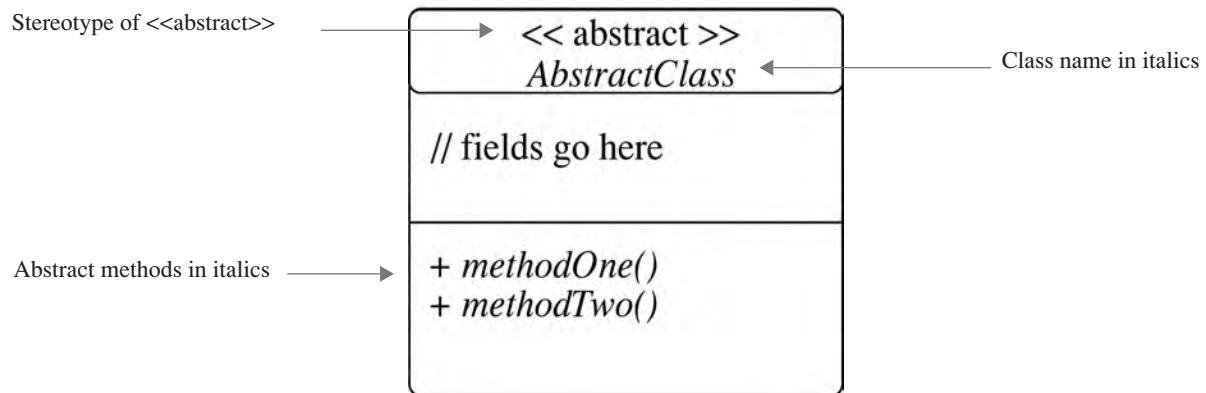


Figure 11-10: Expressing an Abstract Class in the UML

Referring to figure 11-10 — the stereotype `<<abstract>>` is optional but if you draw your UML diagrams by hand it's hard to write in italics, so, it will come in handy. Abstract classes can have fields and concrete methods like normal classes, but abstract methods are shown in italics.

Let's now have a look at a short abstract class inheritance example.

ABSTRACT CLASS EXAMPLE

Figure 11-11 gives the UML class diagram for our example:

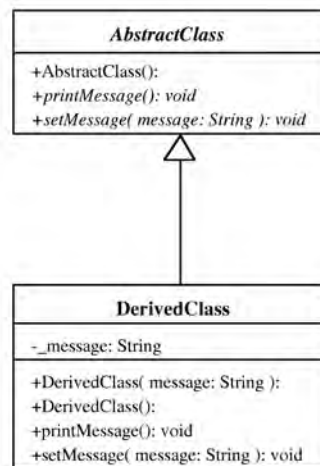


Figure 11-11: UML Class Diagram Showing the `AbstractClass` & `DerivedClass` Inheritance Hierarchy

Referring to figure 11-11 — `AbstractClass` has three methods and no fields. The first method is its default constructor and it is not abstract. (*Constructors cannot be abstract!*) The next two methods, `printMessage()` and `setMessage()` are shown in italics and are therefore abstract methods.

`DerivedClass` inherits from `AbstractClass`. Since `AbstractClass`'s methods are abstract, and have no implementation, `DerivedClass` must provide an implementation for them. `DerivedClass`'s methods are in plain font indicating they have implementations.

Now, if for some reason, you as a designer decided to create a class that inherited from `DerivedClass`, and you defer the implementation of one or more of `DerivedClass`'s methods to that class, then `DerivedClass` would itself have to be declared to be an abstract class. I just wanted to mention this because in most situations you will have more than a two-tiered inheritance hierarchy as I have used here in this simple example.

Let's now take a look at the code for these two classes. Example 11.9 gives the code for `AbstractClass`.

11.9 *AbstractClass.java*

```

1      public abstract class AbstractClass {
2
3          public AbstractClass(){
4              System.out.println("AbstractClass object created!");
5          }
6
7          public abstract void printMessage();
8
9          public abstract void setMessage(String message);
10
11     }
```

The important points to note here is that on line 1 the keyword `abstract` is used to indicate that this is an abstract class definition. The keyword is also used on lines 7 and 9 in the method declarations for `printMessage()` and `setMessage()`. Also note how both the `printMessage()` and `setMessage()` methods are terminated with a semicolon and have no body. (*i.e., no curly braces!*) Example 11.10 gives the code for `DerivedClass`.

11.10 *DerivedClass.java*

```

1      public class DerivedClass extends AbstractClass {
2          private String _message = null;
3
4          public DerivedClass(String message){
5              _message = "Message from DerivedClass: " + message;
6              System.out.println("DerivedClass object created!");
7          }
8
9          public DerivedClass(){
10             this("DerivedClass message!");
11         }
12
13         public void printMessage(){
14             System.out.println(_message);
15         }
16
17         public void setMessage(String message){
18             _message = "Message from DerivedClass: " + message;
19         }
20     }
```

`DerivedClass` extends `AbstractClass` with the `extends` keyword on line 1. Since this version of `AbstractClass` has no `_message` field there is no need for `DerivedClass` to call a base class constructor with the *super* keyword. `DerivedClass` provides an implementation for each of `AbstractClass`'s abstract methods. Let's take a look now at the test driver program that will exercise these two classes.

11.11 *DriverApplication.java*

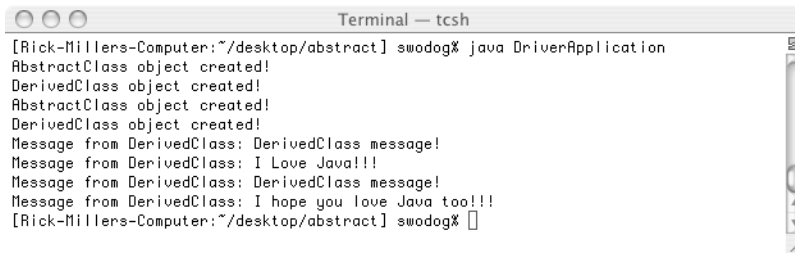
```

1      public class DriverApplication {
2          public static void main(String[] args){
3              AbstractClass ac1 = new DerivedClass();
4              DerivedClass dc1 = new DerivedClass();
5              ac1.printMessage();
6              ac1.setMessage("I Love Java!!!");
7              ac1.printMessage();
8              dc1.printMessage();
9              dc1.setMessage("I hope you love Java too!!!");
10             dc1.printMessage();
11         }
12     }
```

Remember, you cannot directly instantiate an abstract class object. On line 3 a reference to an `AbstractClass` type object named `ac1` is declared and initialized to point to a `DerivedClass` object. On line 4 a reference to a `DerivedClass` type object named `dc1` is declared and initialized to point to a `DerivedClass` object.

Lines 6 through 8 exercises reference `ac1`. The `printMessage()` method is called on line 6 followed by a call to the `setMessage()` method with the String literal “I Love Java!!!” as an argument. The next time the `printMessage()` method is called the new message is printed to the console.

The same series of method calls is performed on the `dc1` reference on lines 10 through 12. Figure 11-12 shows the results of running example 11.11.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/abstract] swodog% java DriverApplication
AbstractClass object created!
DerivedClass object created!
AbstractClass object created!
DerivedClass object created!
Message from DerivedClass: DerivedClass message!
Message from DerivedClass: I Love Java!!!
Message from DerivedClass: DerivedClass message!
Message from DerivedClass: I hope you love Java too!!!
[Rick-Millers-Computer:~/desktop/abstract] swodog% []
```

Figure 11-12: Results of Running Example 11.11

Quick Review

An abstract method is a method that omits its body and has no implementation behavior. A class that declares one or more abstract methods must be declared to be abstract.

The primary purpose of an abstract class is to provide a specification of a set of one or more class interface methods whose implementations are expected to be found in some derived class further down the inheritance hierarchy.

Designers employ abstract classes to provide a measure of application architectural stability.

INTERFACES

An interface is a Java construct that functions like an implicit abstract class. In Java, a derived class can extend the behavior of only one class, but it can implement as many interfaces as it requires.

THE PURPOSE OF INTERFACES

The purpose of an interface is to provide a specification for a set of public interface methods. An interface declaration introduces a new data type, just as a class declaration and definition does.

AUTHORIZED INTERFACE MEMBERS

Java interfaces can only contain four types of members. These include:

- Constant Fields — All interface fields are implicitly public, static, and final
- Abstract Methods — All interface methods are implicitly public and abstract
- Nested Class Declarations — These are implicitly public and static
- Nested Interface Declarations — These have the same characteristics as interfaces!

THE DIFFERENCES BETWEEN AN INTERFACE AND AN ABSTRACT CLASS

Table 11-1 summarizes the differences between abstract classes and interfaces.

Abstract Class	Interface
Must be declared to be abstract with the abstract keyword.	Is implicitly abstract and the use of the abstract keyword is discouraged in an interface declaration.
Can contain abstract as well as concrete methods. The concrete methods can be public, private, or protected.	Can only contain abstract methods. All methods in an interface are implicitly public and abstract. Redundantly declaring interface methods to be public and abstract is permitted but discouraged.
Can contain public, private, and protected variable data fields and constants.	Can only contain public constant fields. All fields in an interface are implicitly public, static, and final. Redundantly declaring an interface field to be public, static, and final is allowed but discouraged.
Can contain nested class and interface declarations.	Can contain nested class and interface declarations. These inner class and interface declarations are implicitly public and static.
Can extend one class but implement many interfaces.	Can extend many interfaces.

Table 11-1: Differences Between Abstract Classes And Interfaces

EXPRESSING INTERFACES IN UML

Interfaces are expressed in the UML in two ways as is shown in figure 11-13.

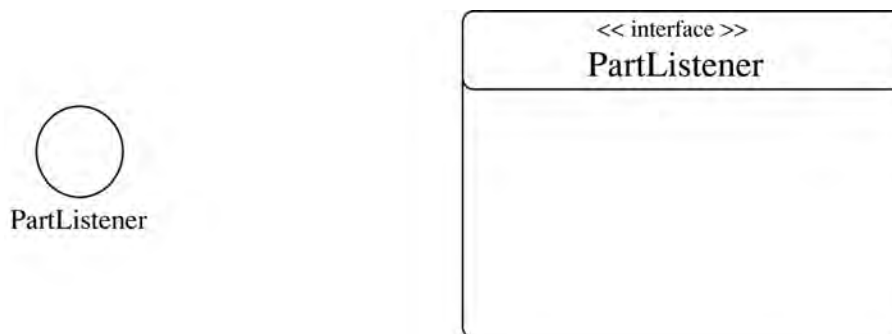


Figure 11-13: Two Types of UML Interface Diagrams

One way to show an interface in the UML is with a simple circle with the name of the interface close by. The second way involves the use of an ordinary class diagram that includes the stereotype << interface >>. Each of these diagrams, the circle and the class diagram, can be used to represent the use of interfaces in an inheritance hierarchy as is discussed in the following section.

EXPRESSING REALIZATION IN A UML CLASS DIAGRAM

When a class implements an interface it is said to be *realizing* the interface. Interface realization is expressed in the UML in two distinct forms: 1) the *simple form* where the circle is used to represent the interface and is combined with an association line to create a *lollipop diagram*, or 2) the *expanded form* where an ordinary class diagram is used to represent the interface. Figure 11-14 illustrates the use of the lollipop diagram to convey the simple form of realization and figure 11-15 shows an example of the expanded form of realization.

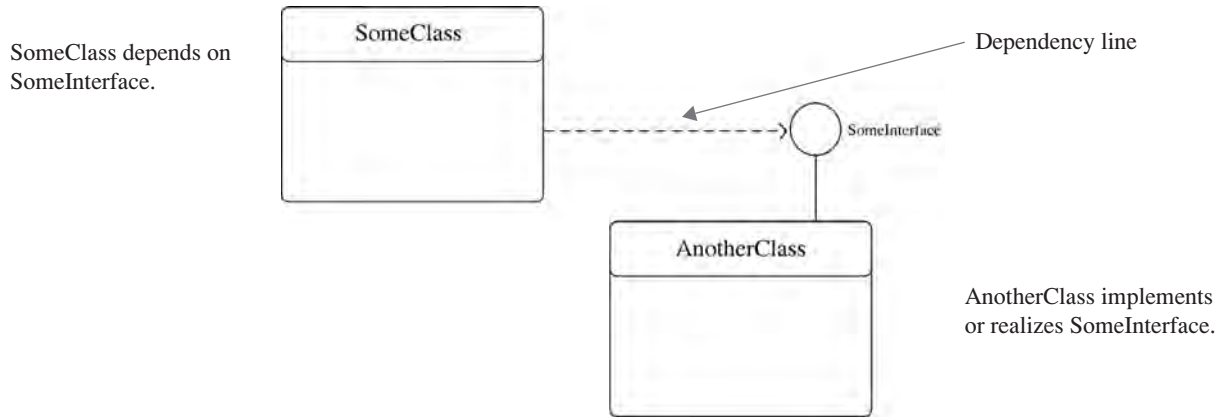


Figure 11-14: UML Diagram Showing the Simple Form of Realization

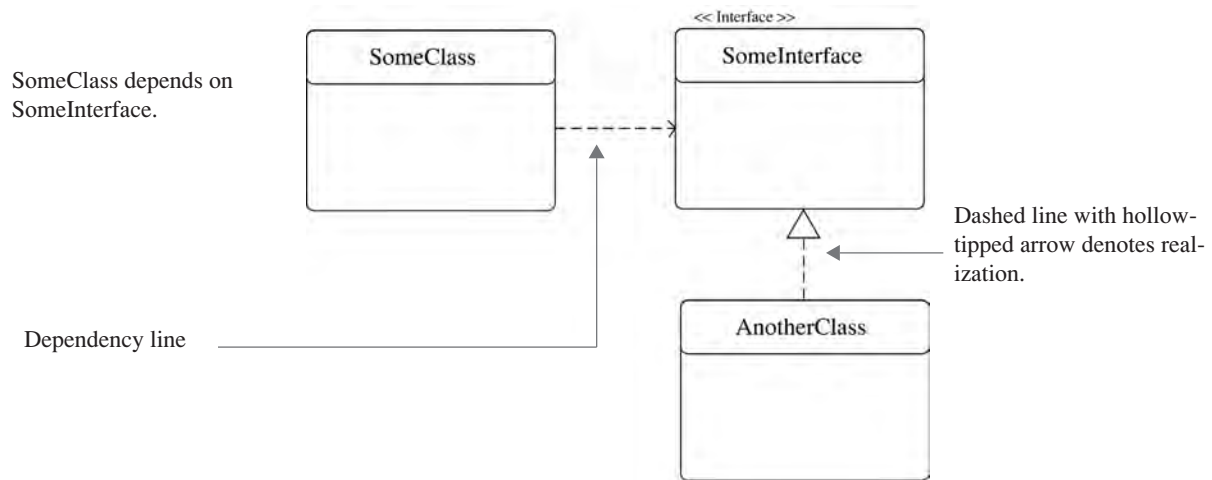


Figure 11-15: UML Diagram Showing the Expanded Form of Realization

AN INTERFACE EXAMPLE

Let's turn our attention to a simple example of an interface in action. Figure 11-16 gives the UML diagram of the interface named `MessagePrinter` and a class named `MessagePrinterClass` that implements the `MessagePrinter` interface. The source code for these two classes is given in examples 11.12 and 11.13.

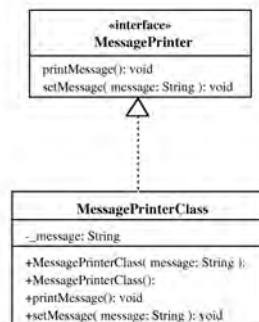


Figure 11-16: UML Diagram Showing the MessagePrinterClass Realizing the MessagePrinter Interface

11.12 MessagePrinter.java

```

1 interface MessagePrinter {
2     void printMessage();
3     void setMessage(String message);
4 }

```

11.13 MessagePrinterClass.java

```

1 public class MessagePrinterClass implements MessagePrinter {
2     private String _message = null;
3
4     public MessagePrinterClass(String message){
5         _message = message;
6     }
7
8     public MessagePrinterClass(){
9         this("Default message is boring!");
10    }
11
12    public void printMessage(){
13        System.out.println(_message);
14    }
15
16    public void setMessage(String message){
17        _message = message;
18    }
19 }

```

11.14 DriverApplication.java

```

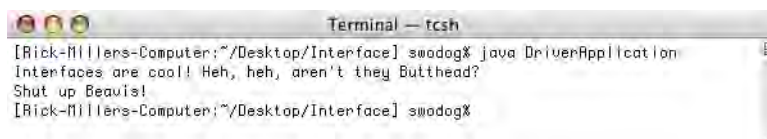
1 public class DriverApplication {
2     public static void main(String[] args){
3         MessagePrinter mp1 = new MessagePrinterClass("Interfaces are cool!" +
4             " Heh, heh, aren't they Butthead?");
5         MessagePrinterClass mpc1 = new MessagePrinterClass("Shut up Beavis!");
6
7         mp1.printMessage();
8         mpc1.printMessage();
9     }
10 }

```

As you can see from example 11.12 the `MessagePrinter` interface is short and simple. All it does is declare the two interface methods `printMessage()` and `setMessage()`. The implementation of these interface methods is left to any class that implements the `MessagePrinter` interface as the `MessagePrinterClass` does in example 11.13.

Example 11.14 gives the test driver program for this example. As you can see on line 3 you can declare an interface type reference. I called this one `mp1`. Although you cannot instantiate an interface directly with the `new` operator you can initialize an interface type reference to point to an object of any concrete class that implements the interface. The only methods you can call on the object via the interface type reference are those methods specified by the interface. You could of course cast to a different type if required but you must strive to minimize the need to cast in this manner.

Figure 11-17 gives the results of running example 11.14.



```

Terminal - tcsh
[Rick-Millers-Computer:~/Desktop/Interface] swadog% java DriverApplication
Interfaces are cool! Heh, heh, aren't they Butthead?
Shut up Beavis!
[Rick-Millers-Computer:~/Desktop/Interface] swadog%

```

Figure 11-17: Results of Running Example 11.14

Quick Review

The purpose of an interface is to specify a set of public interface methods. Interfaces can have four types of members: 1) constants, 2) abstract methods, 3) nested classes, and 4) nested interfaces. Classes can inherit from or extend only one other class, but they can implement as many interfaces as are required. Interfaces can extend as many other interfaces as necessary.

CONTROLLING HORIZONTAL & VERTICAL ACCESS

The term *horizontal access* is used to describe the level of access an object of one type has to the members (*i.e.*, *fields and methods*) of another type. This topic was discussed in detail in chapter 9. The term *vertical access* refers to the level of access a derived class has to its base class members. In both cases access is controlled by the three Java access modifiers *public*, *protected*, and *private*. There is a fourth level of access, known as *package*, that is inferred by the omission of an access modifier. In other words, if you declare a class member, be it a field or method, and don't specify it as being either public, protected, or private, then it is, by default, package. (*The exception to this rule is when you are declaring an interface in which case the members are public by default.*)

The behavior of protected and package accessibility is dictated by what package the classes in question belong. Figure 11-18 gives a diagram showing five classes named ClassOne, ClassTwo, ClassThree, ClassFour, and ClassFive. ClassOne, ClassTwo, and ClassThree belong to Package A, and ClassFour and ClassFive belong to Package B.

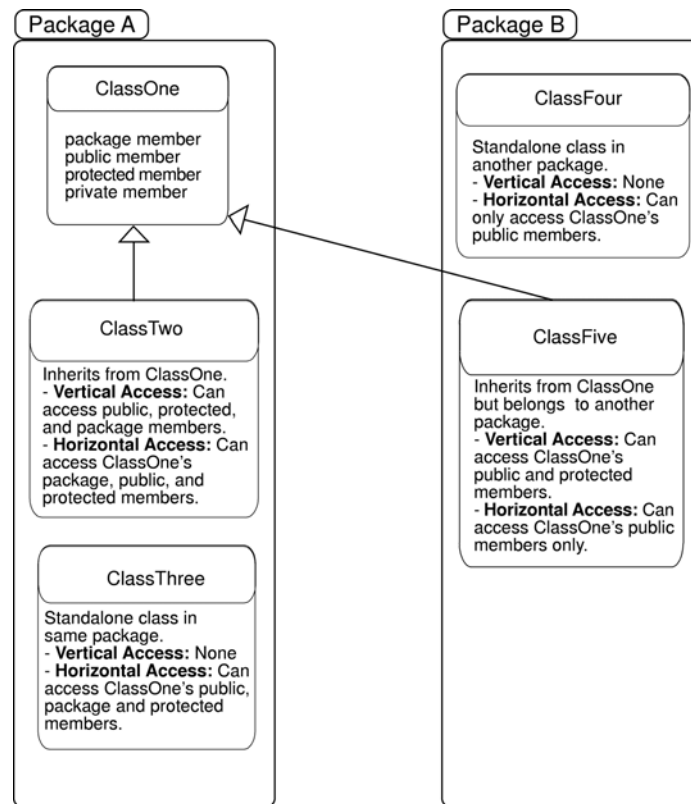


Figure 11-18: Horizontal And Vertical Access In Multi-Package Environment

Referring to figure 11-18 — access behavior is discussed relative to ClassOne from both the horizontal and vertical access perspective. ClassOne contains one package member, one public member, one protected member, and one private member.

ClassTwo inherits from ClassOne and belongs to the same package. From a vertical perspective, as a subclass of ClassOne it has access to ClassOne's public, protected, and package members. From a horizontal perspective it can also access ClassOne's public, protected, and package members.

ClassThree is a standalone class in the same package as ClassOne. Since it is not a subclass of ClassOne it has no vertical access. Horizontally it can access ClassOne's public, protected, and package members.

ClassFour is a standalone class in a different package than ClassOne. It too is not a subclass of ClassOne and therefore has no vertical access. Horizontally it can only access ClassOne's public members.

ClassFive is a subclass of ClassOne but belongs to another package. It has vertical access to ClassOne's public and protected members. It has horizontal access to ClassOne's public members.

The following examples give the code for each class shown in figure 11-18 with offending lines commented out.

11.15 ClassOne.java

```

1     package packageA;
2
3     public class ClassOne {
4         void methodOne(){}
5         public void methodTwo(){}
6         protected void methodThree(){}
7         private void methodFour(){}
8     }

```

11.16 ClassTwo.java

```

1     package packageA;
2
3     public class ClassTwo extends packageA.ClassOne {
4         packageA.ClassOne c1 = new packageA.ClassOne();
5
6         void f(){
7             c1.methodOne();
8             c1.methodTwo();
9             c1.methodThree();
10            // c1.MethodFour(); // Error - no horizontal access to private member
11            methodOne();
12            methodTwo();
13            methodThree();
14            // methodFour(); // Error - no vertical access to private member
15        }
16    }

```

11.17 ClassThree.java

```

1     package packageA;
2
3     public class ClassThree {
4         packageA.ClassOne c1 = new packageA.ClassOne();
5
6         void f(){
7             c1.methodOne();
8             c1.methodTwo();
9             c1.methodThree();
10            // c1.methodFour(); // Error - no horizontal access to private member
11        }
12    }
13

```

11.18 ClassFour.java

```

1     package packageB;
2
3     public class ClassFour {
4         packageA.ClassOne c1 = new packageA.ClassOne();
5
6         void f(){
7             // c1.methodOne(); // Error - no horizontal access to package member in different package
8             c1.methodTwo();
9             // c1.methodThree(); // Error - no horizontal access to protected member
10            // c1.methodFour(); // Error - no horizontal access to private member
11        }
12    }

```

11.19 ClassFive.java

```

1     package packageB;
2
3     public class ClassFive extends packageA.ClassOne {
4         packageA.ClassOne c1 = new packageA.ClassOne();
5
6         void f(){
7             // c1.methodOne(); // Error - no horizontal access to outside package member
8             c1.methodTwo();
9             // c1.methodThree(); // Error - no horizontal access to protected member outside package
10            // c1.MethodFour(); // Error - no horizontal access to private member
11            // methodOne(); // Error - no vertical access outside package
12            methodTwo();
13            methodThree();
14            // methodFour(); // Error - no vertical access - private method
15        }
16    }

```

To test each class simply compile each one. Experiment by removing the comments from an offending line and then compiling and studying the resulting error message(s).

Quick Review

Horizontal and vertical access is controlled via the access specifiers `public`, `protected`, and `private`. A class member without an explicit access modifier declared has package accessibility by default. Essentially, classes belonging to the same package can access each other's `public`, `protected`, and package members horizontally. If a subclass belongs to the same package as its base class it has vertical access to its `public`, `protected`, and package members.

Classes belonging to different packages have horizontal access to each other's `public` members. A subclass in one package whose base class belongs to another package has horizontal access to the base class's `public` methods only but vertical access to both its `public` and `protected` members.

FINAL CLASSES & METHODS

Sometimes you want to prevent classes from being extended or individual methods of a particular class from being overridden. The keyword `final` is used for these purposes. When used to declare a class it prevents that class from being extended. When used to declare a method it prevents the method from being overridden in a derived class.

You cannot use the keyword `final` in combination with the keyword `abstract` for obvious reasons.

Quick Review

Use the `final` keyword to stop the inheritance mechanism or prevent base class methods from being overridden in derived classes.

POLYMORPHIC BEHAVIOR

A good definition of polymorphism is “*The ability to operate on and manipulate different derived objects in a uniform way...*” (Sadr) Add to this the following amplification: “*Without polymorphism, the developer ends up writing code consisting of large case or switch statements. This is in fact the litmus test for polymorphism. The existence of a switch statement that selects an action based upon the type of an object is often a warning sign that the developer has failed to apply polymorphic behavior effectively.*” (Booch)

Polymorphic behavior is easy to understand. In a nutshell it is simply the act of using the set of `public` interface methods defined for a particular class (*or interface*) to interact with that class's (*or interface's*) derived classes. When you write code you need some level of *a priori* knowledge about the type of objects your code will manipulate. In essence, you have to set the bar to some level, meaning that at some point in your code you need to make an assumption about the type of objects with which you are dealing and the behavior they manifest. An object's type, as you know, is important because it specifies the set of operations (*methods*) that are valid for objects of that type (*and subtypes*).

Code that's written to take advantage of polymorphic behavior is generally cleaner, easier to read, easier to maintain, and easier to extend. If you find yourself casting a lot you are not writing polymorphic code. If you use the `instanceof` operator frequently to determine object types then you are not writing polymorphic code. Polymorphic behavior is the essence of object-oriented programming.

Quick Review

Polymorphic behavior is achieved in a program by targeting a set of operations (*methods*) specified by a base class or interface and manipulating their derived class objects via those methods. This uniform treatment of derived

class objects results in cleaner code that's easier to extend and maintain. Polymorphic behavior is the essence of object-oriented programming.

INHERITANCE EXAMPLE: EMPLOYEE

This section offers an inheritance example that extends, once again, the functionality of the Person class given in chapter 9. Figure 11-19 gives the UML diagram.

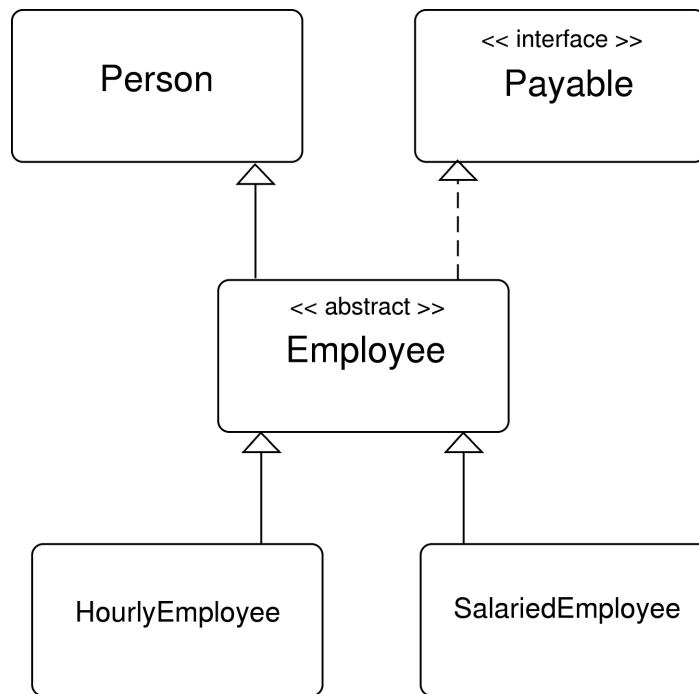


Figure 11-19: Employee Class Inheritance Hierarchy

Referring to figure 11-19 — The Employee class extends Person and implements the Payable interface. However, in this example, as you will see below, the implementation of the method specified in the Payable interface, `pay()`, is deferred by the Employee class to its derived classes. Because the `pay()` method is not actually implemented in the Employee class the Employee class must be declared to be abstract.

HourlyEmployee and SalariedEmployee extend the functionality of Employee. Each of these classes will implement the `pay()` method in their own special way.

From a polymorphic point of view you could write a program that uses these classes in several ways, it just depends on which set of interface methods you want to target. For instance, you could write a program that contains an array of Person references. Each of these Person references could then be initialized to point to an HourlyEmployee object or a SalariedEmployee object. In either case the only methods you can call, without casting, on these objects via a Person reference are those public methods specified by the Person class.

Another approach would be to declare an array of Payable references. Then again you could initialize each Payable reference to point to either an HourlyEmployee object or a SalariedEmployee object. Now the only method you can call on each of these objects, without casting, is the `pay()` method.

A third approach would be to declare an array of Employee references and initialize each reference to point to either an HourlyEmployee object or a SalariedEmployee object. In this scenario you could then call any method specified by Person, Payable, and Employee. This is the approach taken in the EmployeeTestApp program listed below.

The code for each of these classes (*except Person which was shown earlier in the chapter*) is given in examples 11.20 through 11.24.

11.20 Payable.java

```

1 interface Payable {
2     double pay();
3 }

```

11.21 Employee.java

```

1 public abstract class Employee extends Person implements Payable {
2     private String _employee_number = null;
3
4     public Employee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
5                     int dob_day, String gender, String employee_number){
6         super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender);
7         _employee_number = employee_number;
8     }
9
10    public String getEmployeeNumber(){
11        return _employee_number;
12    }
13
14    public String getEmployeeInfo(){
15        return getEmployeeNumber() + " " + getFirstName() + " " + getLastName();
16    }
17 }

```

11.22 HourlyEmployee.java

```

1 public class HourlyEmployee extends Employee {
2     private double _hours_worked;
3     private double _hourly_wage_rate;
4
5     public HourlyEmployee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
6                           int dob_day, String gender, String employee_number,
7                           double hourly_wage_rate, double hours_worked){
8         super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender, employee_number);
9         _hourly_wage_rate = hourly_wage_rate;
10        _hours_worked = hours_worked;
11    }
12
13    public double pay(){
14        return _hourly_wage_rate * _hours_worked;
15    }
16 }

```

11.23 SalariedEmployee.java

```

1 public class SalariedEmployee extends Employee {
2     private double _annual_salary;
3
4     public SalariedEmployee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
5                             int dob_day, String gender, String employee_number, double annual_salary){
6         super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender, employee_number);
7         _annual_salary = annual_salary;
8     }
9
10    public double pay(){
11        return _annual_salary/24; // 24 pay periods
12    }
13 }

```

11.24 EmployeeTestApp.java

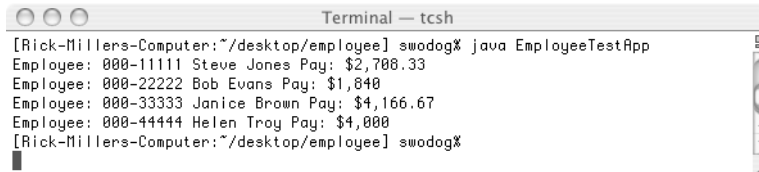
```

1 import java.text.*;
2
3 public class EmployeeTestApp {
4     public static void main(String[] args){
5         Employee[] employees = new Employee[4];
6         NumberFormat number_format = NumberFormat.getInstance();
7         number_format.setMaximumFractionDigits(2);
8
9         employees[0] = new SalariedEmployee("Steve", "J", "Jones", 1983, 3, 4, "M",
10                                           "000-11111", 65000.00);
11        employees[1] = new HourlyEmployee("Bob", "E", "Evans", 1992, 1, 2, "M",
12                                          "000-22222", 23.00, 80.00);
13        employees[2] = new SalariedEmployee("Janice", "A", "Brown", 1983, 3, 4, "F",
14                                          "000-33333", 100000.00);
15        employees[3] = new HourlyEmployee("Helen", "O", "Troy", 1946, 4, 8, "F",
16                                          "000-44444", 50.00, 80.00);
17
18        for(int i = 0; i<employees.length; i++){
19            System.out.println("Employee: " + employees[i].getEmployeeInfo() +
20                               " Pay: " + "$" + number_format.format(employees[i].pay()));
21        }
22    }
23 }

```

Referring to example 11.24 — the `EmployeeTestApp` program declares an array of `Employee` references on line 5 named `employees`. On lines 9 through 16 it initializes each `Employee` reference to point to either an `HourlyEmployee` or `SalariedEmployee` object.

In the `for` statement on line 18 each `Employee` object is manipulated polymorphically via the interface specified by the `Employee` class. (*Which includes the interfaces inherited from `Person` and `Payable`.*) The results of running this program are shown in figure 11-20.



```
Terminal — tcsh
[Rick-Millers-Computer: ~/desktop/employee] swodog% java EmployeeTestApp
Employee: 000-11111 Steve Jones Pay: $2,788.33
Employee: 000-22222 Bob Evans Pay: $1,840
Employee: 000-33333 Janice Brown Pay: $4,166.67
Employee: 000-44444 Helen Troy Pay: $4,800
[Rick-Millers-Computer: ~/desktop/employee] swodog%
```

Figure 11-20: Results of Running Example 11.24

INHERITANCE EXAMPLE: AIRCRAFT ENGINE SIMULATION

As the television chef Emeril Lagasse would say, “Bam! — kick it up a notch!”. This example expands on the aircraft engine simulation originally presented in chapter 10. Here the concepts of inheritance are fused with compositional design to yield a truly powerful combination.

In this example you will also be introduced to the concept of a listener. A listener is a component that responds to events. Listeners are used extensively in the AWT and Swing packages to respond to different GUI events. (*You will be formally introduced to GUI programming in chapter 12.*) This example also utilizes the `Vector` and `Hashtable` collection classes from the `java.util` package and the `synchronized` keyword which is used in several methods in the `Part` class to ensure atomic access to certain objects by client methods. The collection classes are discussed in detail in chapter 17 and the `synchronized` keyword is discussed in detail in chapter 16.

AIRCRAFT ENGINE SIMULATION UML DIAGRAM

Let’s start by discussing the UML diagram for the engine simulation shown in figure 11-21. As you can see from the class diagram this version of the aircraft engine simulation contains decidedly more classes than its chapter 10 cousin. The primary class in the inheritance hierarchy is the `Part` class. Essentially, every class is a `Part` with the exception of the `PartStatus` and `PartEvent` classes. `Part` implicitly inherits from the `java.lang.Object` class but this relationship is not shown on the diagram. The concept of a `Part` encompasses individual as well as composite parts. The `Part` class provides the general functionality for both single as well as composite part objects.

Two other Java API classes that are shown in the class diagram include `EventObject` and `EventListener`. The `PartEvent` class inherits from `EventObject` and the `PartListener` interface inherits from the `EventListener` interface.

The interfaces `IPump`, `ISensor`, and `IEngine` specify the methods that must be implemented by parts of these types. The classes `Pump`, `Sensor`, and `Engine` inherit the basic functionality provided by the `Part` class and implement the appropriate interface.

The `FuelPump` and `OxygenSensor` classes provide concrete implementations for the `Pump` and `Sensor` classes respectively. The `SimpleEngine` class extends `Engine` and implements an interface named `PartListener`. Note here that although `SimpleEngine` implements the `PartListener` interface, other parts could implement the `PartListener` interface as required.

It should also be noted that this program is incomplete at this stage. I have only provided implementations for three concrete classes: `FuelPump`, `OxygenSensor`, and `SimpleEngine`. You will have the opportunity to add parts and functionality to this program in one of the Suggested Projects found at the end of this chapter.

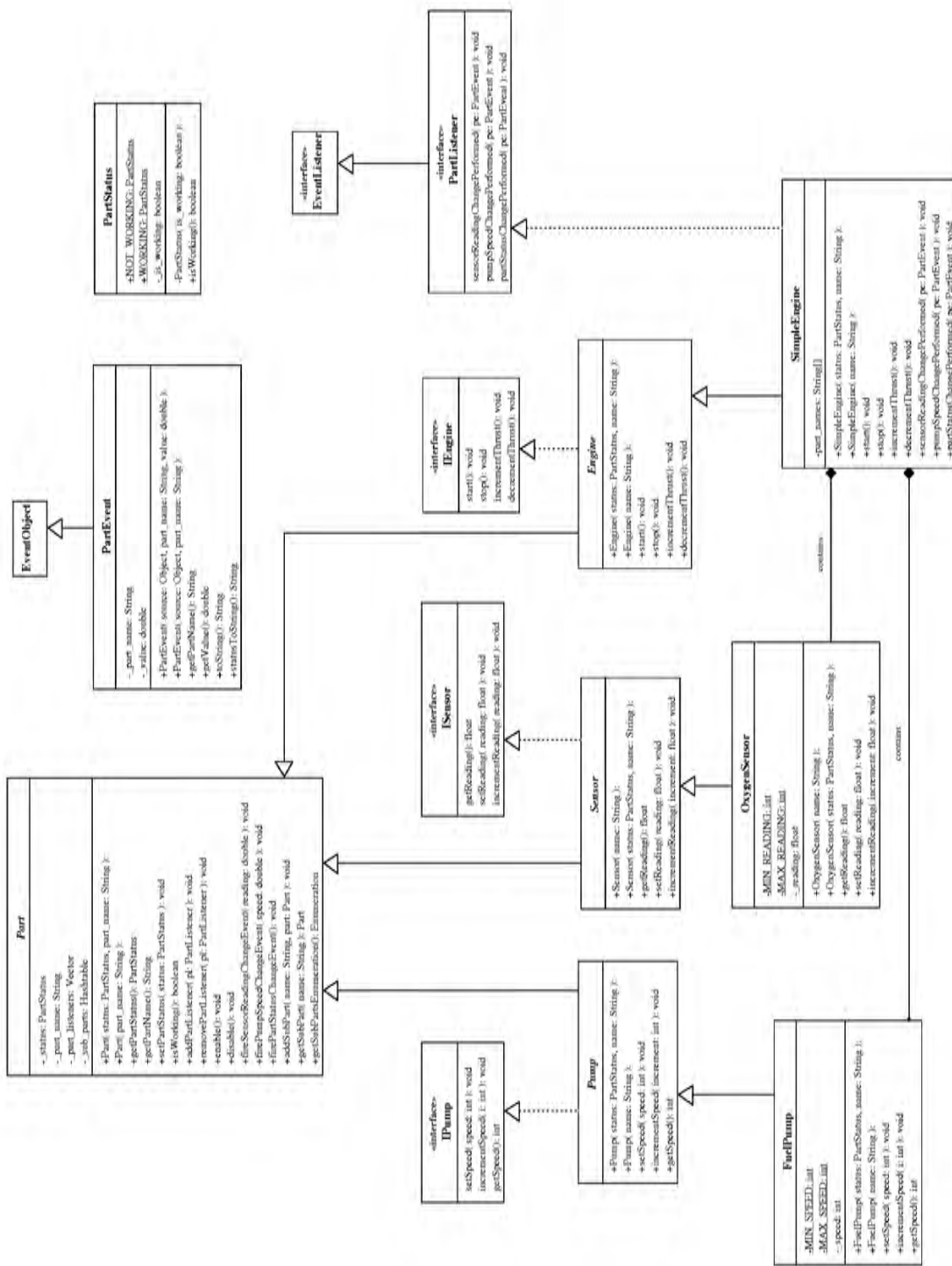


Figure 11-21: UML Class Diagram For Aircraft Engine Simulator

SIMULATION OPERATIONAL DESCRIPTION

The Part class provides the general functionality for all Part objects. A Part object can be a single, atomic part or it can be an aggregate of several Part objects. The Parts comprising a composite Part object are stored in a Part's `_sub_parts` Hashtable. Part objects are added to the `_sub_parts` Hashtable by calling the `addSubPart()` method.

A particular subclass of Part may be a PartListener if it implements the PartListener interface. The PartListener interface specifies three methods that correspond to state changes on Sensors, Pumps, and the working status of a part. A Part that is also a PartListener will need to register itself with the Part of interest. Take for example the SimpleEngine class. Referring to the code listing given in example 11.35 — on lines 9 and 10 in the constructor, the SimpleEngine class creates two subparts and adds them to the `_sub_parts` Hashtable by calling the `addSubPart()` method. On lines 11 and 12 it then gets a reference to each of its subparts by calling the `getSubPart()` method and registering itself as a PartListener for each of its subparts.

To see how the PartListener mechanism works let's step through a pump speed change. In the `start()` method of the SimpleEngine class the speed of its FuelPump subpart is set to 100. (*see line 31 of example 11.35*) The `setSpeed()` method is called on a Pump interface but, because the FuelPump class overrides Pump's `setSpeed()` method, it is the FuelPump's version of `setSpeed()` that gets called. Now, go to the FuelPump class listed in example 11.33 and find the `setSpeed()` method. If the speed desired falls within the valid range of a FuelPump object then the Pump version of `setSpeed()` is called via the *super* object. Go now to the Pump class listed in example 11.30 and find its version of the `setSpeed()` method. It's sole job is to call the `firePumpSpeedChangeEvent()` method with the indicated speed as an argument. The `firePumpSpeedChangeEvent()` method is located in the Part class so go now to example 11.25 and find it.

The `firePumpSpeedChangeEvent()` does several things. First, it checks to see if its `_part_listeners` Vector is empty. If it is then it doesn't have any interested PartListeners. If it has one or more interested PartListeners it clones its `_part_listeners` Vector object in a synchronized block to prevent sporadic behavior, creates a new PartEvent object, converts the `listeners_copy` vector to an Enumeration, and then steps through each element of the Enumeration calling the `pumpSpeedChangePerformed()` method on each interested PartListener object. This is when the `pumpSpeedChangePerformed()` method is called on the SimpleEngine object resulting in the console output you see when you run the program.

TAKE A DEEP BREATH AND RELAX!

At this point, since you have yet to be formally introduced to them, some of the concepts regarding the use of Vectors, Hashtables, and Enumerations will seem cryptic. The PartListener firing mechanism is also difficult at first to understand. If you don't fully grasp the complete workings of this complex example right now don't worry. My intent is to challenge you to dive into the code and trace the execution. That's how you learn how to program!

COMPILING THE AIRCRAFT ENGINE SIMULATION CODE

The easiest way to compile all the aircraft engine simulation code is to put the source files in one directory and issue the following command:

```
javac -source 1.4 *.java
```

You need to add the `-source 1.4` option because of the use of the `assert` keyword in the Part class.

COMPLETE AIRCRAFT ENGINE SIMULATION CODE LISTING

11.25 Part.java

```

1      import java.util.*;
2
3      public abstract class Part {
4
5          /*****
6              private attributes
7          *****/
8          private PartStatus _status = null;
9          private String _part_name = null;
```

```

10     private Vector _part_listeners = null;
11     private Hashtable _sub_parts = null;
12
13
14     /*****
15     constructor
16     *****/
17     public Part(PartStatus status, String part_name){
18         _status = status;
19         _part_name = part_name;
20         System.out.println("Part object created!");
21     }
22
23     /*****
24     default constructor
25     *****/
26     public Part(String part_name){
27         _status = PartStatus.WORKING;
28         _part_name = part_name;
29         System.out.println("Part object created!");
30     }
31
32     /*****
33     public interface methods
34     *****/
35
36     public PartStatus getPartStatus(){ return _status; }
37
38     public String getPartName(){ return _part_name; }
39
40     public void setPartStatus(PartStatus status){
41         _status = status;
42         firePartStatusChangeEvent();
43     }
44
45     public boolean isWorking(){
46         if(_sub_parts != null){ // there are subparts
47             for(Enumeration e = getSubPartsEnumeration(); e.hasMoreElements();){
48                 if(((Part) e.nextElement()).isWorking()) {
49                     continue; //great!
50                 }else{
51                     setPartStatus(PartStatus.NOT_WORKING);
52                     break;
53                 }
54             }
55         }
56         System.out.println(getPartName() + " " + " isWorking status = " + _status.isWorking());
57         return _status.isWorking();
58     }
59
60     public void addPartListener(PartListener pl){
61         if(_part_listeners == null){
62             _part_listeners = new Vector();
63             _part_listeners.add(pl);
64         }else {
65             _part_listeners.add(pl);
66         }
67     }
68
69     public void removePartListener(PartListener pl){
70         if(_part_listeners == null){
71             ; //do nothing
72         } else {
73             if(_part_listeners.isEmpty()){
74                 ; //do nothing
75             } else {
76                 _part_listeners.removeElement(pl);
77             }
78         }
79     }
80
81     public void enable(){
82         setPartStatus(PartStatus.WORKING);
83     }
84
85     public void disable(){
86         setPartStatus(PartStatus.NOT_WORKING);
87     }
88
89
90     public void fireSensorReadingChangeEvent(double reading){

```

```

91     if(_part_listeners.isEmpty()){
92         ; //there's nothing to do!
93     } else {
94         Vector listeners_copy = null;
95         synchronized(this){
96             listeners_copy = (Vector)_part_listeners.clone();
97         }
98         PartEvent pe = new PartEvent(this, _part_name, reading);
99         for(Enumeration e = listeners_copy.elements(); e.hasMoreElements();){
100             ((PartListener) (e.nextElement())).sensorReadingChangePerformed(pe);
101         }
102     }
103 }
104
105 public void firePumpSpeedChangeEvent(double speed){
106     if(_part_listeners.isEmpty()){
107         ; //there's nothing to do!
108     } else {
109         Vector listeners_copy = null;
110         synchronized(this){
111             listeners_copy = (Vector)_part_listeners.clone();
112         }
113         PartEvent pe = new PartEvent(this, _part_name, speed);
114         for(Enumeration e = listeners_copy.elements(); e.hasMoreElements();){
115             ((PartListener) (e.nextElement())).pumpSpeedChangePerformed(pe);
116         }
117     }
118 }
119
120 public void firePartStatusChangeEvent(){
121     if(_part_listeners.isEmpty()){
122         ; //there's nothing to do!
123     } else {
124         Vector listeners_copy = null;
125         synchronized(this){
126             listeners_copy = (Vector)_part_listeners.clone();
127         }
128         PartEvent pe = new PartEvent(this, _part_name);
129         for(Enumeration e = listeners_copy.elements(); e.hasMoreElements();){
130             ((PartListener) (e.nextElement())).partStatusChangePerformed(pe);
131         }
132     }
133 }
134
135 public void addSubPart(String name, Part part){
136     if(_sub_parts == null){
137         _sub_parts = new Hashtable();
138         _sub_parts.put(name, part);
139     }else{
140         _sub_parts.put(name, part);
141     }
142 }
143
144 public Part getSubPart(String name){
145     assert (_sub_parts != null); //can't call if no subparts exist!
146     return (Part)_sub_parts.get(name);
147 }
148
149 public Enumeration getSubPartsEnumeration(){
150     assert (_sub_parts != null );
151     return _sub_parts.elements();
152 }
153 }

```

11.26 IPump.java

```

1 public interface IPump {
2     public void setSpeed(int speed);
3     public void incrementSpeed(int i);
4     public int getSpeed();
5 }

```

11.27 ISensor.java

```

1 public interface ISensor {
2     public float getReading();
3     public void setReading(float reading);
4     public void incrementReading(float reading);
5 }

```

11.28 IEngine.java

```

1     public interface IEngine {
2         public void start();
3         public void stop();
4         public void incrementThrust();
5         public void decrementThrust();
6     }

```

11.29 PartListener.java

```

1     import java.util.*;
2
3     public interface PartListener extends EventListener {
4         public void sensorReadingChangePerformed(PartEvent pe);
5         public void pumpSpeedChangePerformed(PartEvent pe);
6         public void partStatusChangePerformed(PartEvent pe);
7     }

```

11.30 Pump.java

```

1     public abstract class Pump extends Part implements IPump {
2
3         public Pump(PartStatus status, String name){
4             super(status, name);
5             System.out.println("Pump object created!");
6         }
7
8         public Pump(String name){
9             super(name);
10            System.out.println("Pump object created!");
11        }
12
13        public void setSpeed(int speed){
14            firePumpSpeedChangeEvent(speed);
15        }
16
17        public void incrementSpeed(int increment){
18            firePumpSpeedChangeEvent(getSpeed());
19        }
20
21        public int getSpeed() { return 0; }
22    }
23

```

11.31 Sensor.java

```

1     public abstract class Sensor extends Part implements ISensor {
2
3         public Sensor(String name){
4             super(name);
5             System.out.println("Sensor object created!");
6         }
7
8         public Sensor(PartStatus status, String name){
9             super(status, name);
10            System.out.println("Sensor object created!");
11        }
12
13
14        /*****
15         ISensor Interface Methods
16         *****/
17
18        public float getReading(){
19            return 0;
20        }
21
22        public void setReading(float reading){
23            fireSensorReadingChangeEvent(reading);
24        }
25
26        public void incrementReading(float increment) {
27            fireSensorReadingChangeEvent(getReading());
28        }
29    }

```

11.32 Engine.java

```

1      public abstract class Engine extends Part implements IEngine {
2
3          public Engine(PartStatus status, String name){
4              super(status, name);
5              System.out.println("Engine object created!");
6          }
7
8          public Engine(String name){
9              super(name);
10             System.out.println("Engine object created!");
11         }
12
13         /*****
14             IEngine Interface Method Stubs
15
16             Default behavior will do nothing. They must be
17             overridden in a derived class.
18             *****/
19         public void start(){ }
20         public void stop(){ }
21         public void incrementThrust(){ }
22         public void decrementThrust(){ }
23     }
24

```

11.33 FuelPump.java

```

1      public class FuelPump extends Pump{
2
3          private static final int MIN_SPEED = 0;
4          private static final int MAX_SPEED = 1000;
5
6          private int _speed = 0;
7
8          public FuelPump(PartStatus status, String name){
9              super(status, name);
10             System.out.println("Fuel Pump " + name + " created!");
11         }
12
13         public FuelPump(String name){
14             super(name);
15             System.out.println("Fuel Pump " + name + " created!");
16         }
17
18         public void setSpeed(int speed){
19             if((speed >= MIN_SPEED) && (speed <= MAX_SPEED)){
20                 _speed = speed;
21                 super.setSpeed(speed);
22             }else{
23                 System.out.println("Pump speed cannot exceed specified range:" +
24                     MIN_SPEED + " - " + MAX_SPEED);
25             }
26         }
27
28         public void incrementSpeed(int i){
29             if(((_speed + i) >= MIN_SPEED) && ((_speed + i) <= MAX_SPEED)){
30                 _speed += i;
31                 super.incrementSpeed(i);
32             } else {
33                 System.out.println("Pump speed cannot exceed specified range: " + MIN_SPEED + " - " + MAX_SPEED);
34             }
35         }
36
37         public int getSpeed() { return _speed; }
38     }
39

```

11.34 OxygenSensor.java

```

1      public class OxygenSensor extends Sensor {
2          private static final int MIN_READING = 0;
3          private static final int MAX_READING = 1000;
4          private float _reading = 0;
5
6          public OxygenSensor(String name){
7              super(name);
8              System.out.println("Oxygen Sensor " + name + " object created!");
9              _reading = 0;

```

```

10     }
11
12     public OxygenSensor(PartStatus status, String name){
13         super(status, name);
14         System.out.println("Oxygen Sensor " + name + " object created!");
15         _reading = 0;
16     }
17
18
19     public float getReading(){ return _reading; }
20
21     public void setReading(float reading){
22         if( (reading >= MIN_READING) && (reading <= MAX_READING)){
23             _reading = reading;
24             super.setReading(reading); //call Sensor method to fire sensor reading change event
25         } else {
26             System.out.println("Warning: Attempt to set Oxygen Sensor outside valid range: " +
27                 MIN_READING + " - " + MAX_READING);
28         }
29     }
30 }
31
32 public void incrementReading(float increment){
33     if(((_reading + increment) >= MIN_READING) && ((_reading + increment) <= MAX_READING)){
34         _reading += increment;
35         super.setReading(_reading); //call Sensor method to fire sensor reading change event
36     } else {
37         System.out.println("Warning: Attempt to set Oxygen Sensor outside valid range: " +
38             MIN_READING + " - " + MAX_READING);
39     }
40 }
41 } // end OxygenSensor class
42

```

11.35 SimpleEngine.java

```

1     public class SimpleEngine extends Engine implements PartListener {
2
3
4         private String[] part_names = {"FuelPump 1", "Oxygen Sensor 1"};
5
6         public SimpleEngine(PartStatus status, String name){
7             super(status, name);
8
9             addSubPart(part_names[0], new FuelPump(part_names[0]));
10            addSubPart(part_names[1], new OxygenSensor(part_names[1]));
11            getSubPart(part_names[0]).addPartListener(this);
12            getSubPart(part_names[1]).addPartListener(this);
13            System.out.println("SimpleEngine object created!");
14        }
15
16        public SimpleEngine(String name){
17            super(name);
18
19            addSubPart(part_names[0], new FuelPump(part_names[0]));
20            addSubPart(part_names[1], new OxygenSensor(part_names[1]));
21            getSubPart(part_names[0]).addPartListener(this);
22            getSubPart(part_names[1]).addPartListener(this);
23            System.out.println("SimpleEngine object created!");
24        }
25
26        /*****
27         IEngine Interface Method Implementations
28         *****/
29        public void start(){
30            if(isWorking()){
31                ((Pump) getSubPart(part_names[0])).setSpeed(100);
32            }
33        }
34
35
36        public void stop(){
37            ((Pump) getSubPart(part_names[0])).setSpeed(0);
38        }
39
40        public void incrementThrust(){
41            ((Pump) getSubPart(part_names[0])).incrementSpeed(25);
42        }
43
44        public void decrementThrust(){
45

```

```

46         ((Pump) getSubPart(part_names[0])).incrementSpeed(-25);
47     }
48
49
50     /*****
51     PartListener Interface Methods
52     *****/
53
54     public void sensorReadingChangePerformed(PartEvent pe){
55         System.out.println("SimpleEngine: sensorReadingChangePerformed() method called!");
56         System.out.println(pe.getPartName() + " reading changed to " + pe.getValue());
57     }
58
59
60     public void pumpSpeedChangePerformed(PartEvent pe){
61         System.out.println("SimpleEngine: pumpSpeedChangePerformed() method called!");
62         System.out.println(pe.getPartName() + " speed changed to " + pe.getValue());
63     }
64
65     public void partStatusChangePerformed(PartEvent pe){
66         System.out.println("SimpleEngine: partStatusChangePerformed() method called!");
67         System.out.println(pe.getPartName() + pe.statusToString());
68     }
69
70 }

```

11.36 PartEvent.java

```

1     import java.util.*;
2
3     public class PartEvent extends EventObject {
4         private String _part_name = null;
5         private double _value = 0.0;
6
7         public PartEvent(Object source, String part_name, double value){
8             super(source);
9             _part_name = part_name;
10            _value = value;
11        }
12
13        public PartEvent(Object source, String part_name){
14            super(source);
15            _part_name = part_name;
16        }
17
18        public String getPartName(){ return _part_name; }
19
20        public double getValue(){ return _value; }
21
22        public String toString(){ return super.toString() + " " + getPartName();}
23
24        public String statusToString(){
25            return ((Part) source).isWorking() ? " is working properly!" : " has malfunctioned!";
26        }
27
28    }

```

11.37 PartStatus.java

```

1     public final class PartStatus {
2         public static final PartStatus NOT_WORKING = new PartStatus(false);
3         public static final PartStatus WORKING = new PartStatus(true);
4
5         private boolean _is_working = false;
6
7         private PartStatus(boolean is_working){
8             _is_working = is_working;
9         }
10
11        public boolean isWorking(){ return _is_working; }
12    }

```

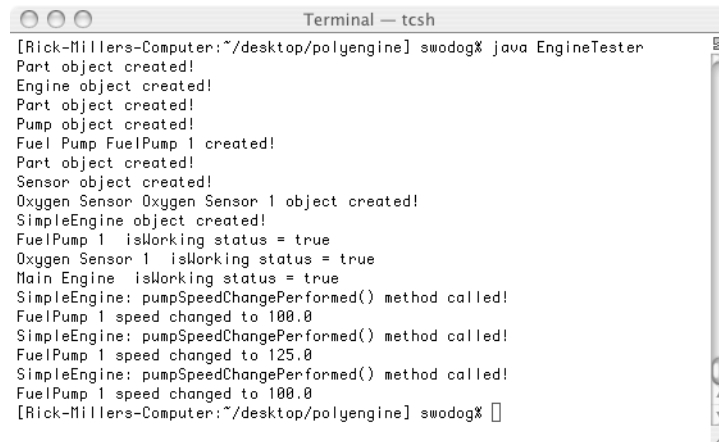
11.38 EngineTester.java

```

1     public class EngineTester {
2         public static void main(String[] args){
3             IEngine engine = new SimpleEngine("Main Engine");
4
5             engine.start();
6             engine.incrementThrust();
7             engine.decrementThrust();
8         }
9     }

```

Figure 11-22 shows the results of running example 11.38.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/polyengine] swodog% java EngineTester
Part object created!
Engine object created!
Part object created!
Pump object created!
Fuel Pump FuelPump 1 created!
Part object created!
Sensor object created!
Oxygen Sensor Oxygen Sensor 1 object created!
SimpleEngine object created!
FuelPump 1 isWorking status = true
Oxygen Sensor 1 isWorking status = true
Main Engine isWorking status = true
SimpleEngine: pumpSpeedChangePerformed() method called!
FuelPump 1 speed changed to 100.0
SimpleEngine: pumpSpeedChangePerformed() method called!
FuelPump 1 speed changed to 125.0
SimpleEngine: pumpSpeedChangePerformed() method called!
FuelPump 1 speed changed to 100.0
[Rick-Millers-Computer:~/desktop/polyengine] swodog%

```

Figure 11-22: Results of Running Example 11.38

TERMS & DEFINITIONS

Term	Definition
class	A Java construct that introduces a new data type. A class definition can have package, public, protected, or private members. Class methods can implement behavior.
type	The specification of a set of valid operations over a set of objects. You can only perform an operation on an object if the operation is supported by the object's type.
interface	A Java construct that introduces a new data type. Interface methods only specify behavior — they have no implementation.
base class	A class whose behavior is inherited (<i>extended</i>) by a derived class.
derived class	A class that inherits (<i>extends</i>) the behavior of an existing class. A derived class can extend only one class but can implement as many interfaces as necessary.
abstract method	A class method that omits its body and therefore has no implementation. Interface methods are abstract by default.
abstract class	A class that declares one or more abstract methods. Also, a class that implements an interface but defers implementing the interface's method(s) to a derived class further down the inheritance hierarchy.

Table 11-2: Chapter 11 Terms and Definitions

SUMMARY

Inheritance serves three essential purposes: 1) it is an object-oriented design mechanism that enables you to think and reason about your program structure in terms of generalized and specialized class behavior, 2) it provides you with a measure of code reuse within your program by locating common class behavior in base classes, and 3) it serves as a means to incrementally develop programs over time.

Classes that belong to an inheritance hierarchy participate in an “is a” relationship between themselves and their chain of base classes. This “is a” relationship is transitive in the direction of specialized to generalized classes but not vice versa.

The Java class and interface constructs are each used to create new, user-defined data types. The interface construct is used to specify a set of authorized type methods and omits method behavior; the class construct is used to specify a set of authorized type methods and their behavior. A class construct, like an interface, can omit the bodies of one or more of its methods, however, such methods must be declared to be abstract. A class that declares one or more of its methods to be abstract must itself be declared to be an abstract class. Abstract class objects cannot be created with the new operator.

A base class implements default behavior in the form of methods that can be inherited by derived classes. There are three reference -> object combinations: 1) if the base class is a concrete class, meaning it is not abstract, then a base class reference can point to a base class object, 2) a base class reference can point to a derived class object, and 3) a derived class reference can point to a derived class object.

Reference variables have an associated type. Method calls to an object pointed to by a reference will succeed without casting so long as the type of the reference supports the method you are trying to call. You can force, or coerce, the compiler to treat a reference to an object of one type as if it were a reference to an object of another. This is extremely helpful in some circumstances but, as a rule, casting should be used sparingly. Also, casting only works if the object really is of the type you are casting it to.

Derived classes can override base class behavior by providing overriding methods. An overriding method is a method in a derived class that has the same method signature as the base class method it is intending to override. Overriding methods can be called polymorphically via a base class reference that points to a derived class object.

An abstract method is a method that omits its body and has no implementation behavior. A class that declares one or more abstract methods must be declared to be abstract.

The primary purpose of an abstract class is to provide a specification of a set of one or more class interface methods whose implementations are expected to be found in some derived class further down the inheritance hierarchy.

Designers employ abstract classes to provide a measure of application architectural stability.

The purpose of an interface is to specify a set of public interface methods. Interfaces can have four types of members: 1) constants, 2) abstract methods, 3) nested classes, and 4) nested interfaces. Classes can inherit from or extend only one other class, but they can implement as many interfaces as are required. Interfaces can extend as many other interfaces as necessary.

Horizontal and vertical access is controlled via the access specifiers public, protected, and private. A class member without an explicit access modifier declared has package accessibility by default. Essentially, classes belonging to the same package can access each other’s public, protected, and package members horizontally. If a subclass belongs to the same package as its base class it has vertical access to its public, protected, and package members.

Classes belonging to different packages have horizontal access to each other’s public members. A subclass in one package whose base class belongs to another package has horizontal access to the base class’s public methods only but vertical access to both its public and protected members.

Use the *final* keyword to stop the inheritance mechanism or prevent base class methods from being overridden in derived classes.

Polymorphic behavior is achieved in a program by targeting a set of operations (*methods*) specified by a base class or interface and manipulating their derived class objects via those methods. This uniform treatment of derived class objects results in cleaner code that’s easier to extend and maintain. Polymorphic behavior is the essence of object-oriented programming.

Skill-Building Exercises

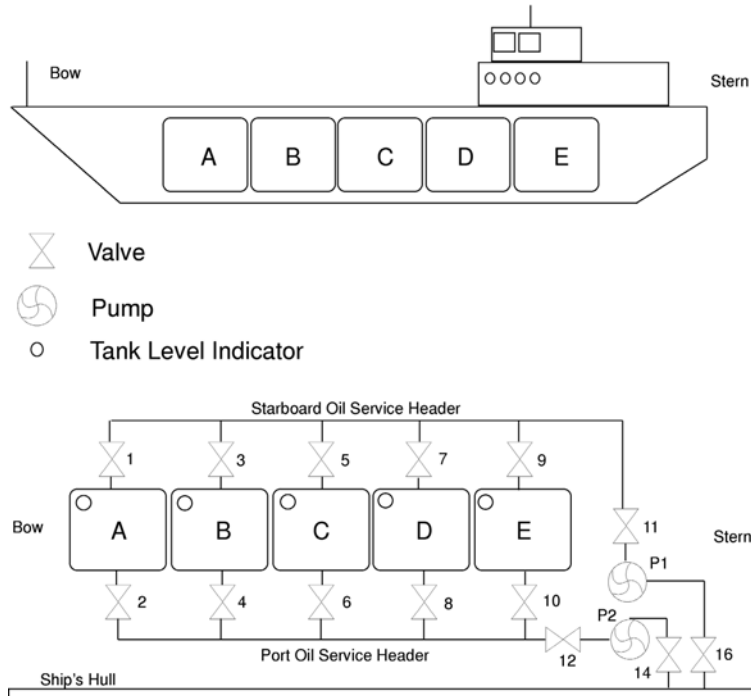
1. **Question:** How many classes can a derived class extend? How many interfaces can a derived class implement?
2. **Question:** How many interfaces can a derived interface extend?

3. **API Familiarization:** Look up the Vector, Hashtable, and Enumeration classes in the Java API documentation and study their usage. This will help you to understand their usage in the aircraft engine simulation code presented in this chapter.
4. **Simple Inheritance:** Write a small program to test the effects of inheritance. Create a class named ClassA that implements the following methods: a(), b(), and c(). Each method should print a short text message to the screen. Create a default constructor for ClassA that prints a message to the screen announcing the creation of a ClassA object. Next, create a class named ClassB that extends ClassA. Give ClassB a default constructor that announces the creation of a ClassB object. In a test driver program create three references. Two of the references should be of type ClassA and the third should be of type ClassB. Initialize the first reference to point to a ClassA object, initialize the second reference to point to a ClassB object, and initialize the third reference to point to a ClassB object as well. Via each of the references call the methods a(), b(), and c(). Run the test driver program and note the results.
5. **Overriding Methods:** Reusing some of the code you created in the previous exercise create another class named ClassC that extends ClassA and provides overriding methods for each of ClassA's methods a(), b(), and c(). Have each of the methods defined in ClassC print short messages to the screen. In the test driver program declare three references, the first two of type ClassA and the third of type ClassC. Initialize the first reference to point to an object of type ClassA, the second to point to an object of type ClassC, and the third to point to an object of ClassC as well. Via each of the references call the methods a(), b(), and c(). Run the test driver program and note the results.
6. **Abstract Classes:** Create an abstract class named AbstractClassA and give it a default constructor and three abstract methods named a(), b(), and c(). Create another class named ClassB that extends ClassA. Provide overriding methods for each of the abstract methods declared in ClassA. Each overriding method should print a short text message to the screen. Create a test driver program that declares two references. The first reference should be of type ClassA, the second reference should be of type ClassB. Initialize the first reference to point to an object of type ClassB, and the second reference to point to an object of ClassB as well. Via each reference call the methods a(), b(), and c(). Run the program and note the results.
7. **Interfaces:** Convert the abstract class you created in the previous exercise to an interface. What changes did you have to make to the code? Compile your interface and test driver program code, re-run the program, and note the results.
8. **Mental Exercise:** Consider the scenario then answer the questions that follow: Given an abstract base class named ClassOne with the following abstract public interface methods a(), b(), and c(). Given a class named ClassTwo that derives from ClassOne, provides implementations for each of ClassOne's abstract methods, and defines one additional method named d(). Now, you have two references. One is of type ClassOne, the other of type ClassTwo. What methods can be called via the ClassOne reference without casting? Likewise, what methods can be called via the ClassTwo reference without casting?

SUGGESTED PROJECTS

1. **Draw Sequence Diagram:** Draw a UML sequence diagram of the SimpleEngine constructor call. Refer to the code supplied in examples 11.25 through 11.38.
2. **Draw Sequence Diagram:** Draw a UML sequence diagram of the SimpleEngine start() method.
3. **Draw Sequence Diagram:** Draw a UML sequence diagram of the SimpleEngine incrementThrust() method.

- Extend Functionality:** Extend the functionality of the Employee example given in this chapter. Create a subclass named PartTimeEmployee that extends HourlyEmployee. Limit the number of hours a PartTimeEmployee can have to 30 hours per pay period.
- Extend Functionality:** Extend the functionality of the aircraft engine simulation program given in this chapter. Create several more types of pumps and sensors. For instance, you might create an OilPump or an OilTemperature-Sensor class. What other types of parts might you want to model in the simulation? Create different types of engines that consist of different types of parts. Use your imagination!
- Oil Tanker Pumping System:** Design and create an oil tanker pumping system simulation. Assume your tanker ship has five oil cargo compartments as shown in the diagram below.



Each compartment can be filled and drained from either the port or starboard service header. The oil pumping system consists of 14 valves numbered 1 through 16. Even-numbered valves are located on the port side of the ship and odd-numbered valves are located on the starboard side of the ship. (*Valve numbers 13 and 15 are not used.*)

The system also consists of two pumps that can be run in two speeds, slow or fast speed, and in two directions, drain and fill. When a pump is running in the drain direction it is taking a suction from the tank side, when running in the fill direction it is taking a suction from the hull side. Assume a pumping capacity of 1000 gallons per minute in fast mode.

Each tank contains one tank level indicator that is a type of sensor. The indicators sense a continuous tank level from 0 (empty) to 100,000 gallons.

Your program should let you drain and fill the oil compartments by opening and closing valves and starting and setting pump speeds. For instance, to fill tank A quickly you could open valves 1, 2, 11, 12, 14 & 16, and start pumps P1 and P2 in the fill direction in the fast mode.

SELF-TEST QUESTIONS

1. What are the three essential purposes of inheritance?

2. A class that belongs to an inheritance hierarchy participates in what type of relationship with its base class?
3. Describe the relationship between the terms interface, class, and type.
4. How do you express generalization and specialization in a UML class diagram? You may draw a picture to answer the question.
5. Describe how to override a base class method in a derived class.
6. Why would it be desirable to override a base class method in a derived class?
7. What's the difference between an ordinary method and an abstract method?
8. How many abstract methods must a class have before it must be declared to be an abstract class?
9. List several differences between classes and interfaces.
10. How do you express an abstract class in a UML class diagram?
11. Hi, I'm a class that declares a set of interface methods but fails to provide an implementation for the those methods. What type of class am I?
12. List the four authorized members of an interface.
13. What two ways can you express realization in a UML class diagram. You may use pictures to answer the question.
14. How do you call a base class constructor from a derived class constructor?
15. How do you call a base class method, other than a constructor, from a derived class method?
16. Describe the effects using the access modifiers public, package, private, and the default access package, has on classes belonging to the same and different packages. State the effects from the both the horizontal and vertical member access perspectives.
17. What can you do to prevent or stop a class from being inherited?
18. What can you do to prevent a method from being overridden in a derived class?
19. State, in your own words, a good definition for the term polymorphism.

REFERENCES

Grady Booch, et. al. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1998. ISBN: 0-201-57168-4

Rick Miller. *C++ For Artists: The Art, Philosophy, And Science of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA, 2003. ISBN: 1-932504-02-8

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Second Edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994. ISBN: 0-8053-5340-2

Babak Sadr. *Unified Objects: Object-Oriented Programming Using C++*. The IEEE Computer Society, Los Alamitos, CA. ISBN: 0-8186-7733-3

Antero Taivalsaari. *On the Notion of Inheritance*. ACM Computing Surveys, Vol. 28, No. 3, September 1996, pp. 438 - 479.

Clyde Ruby and Gary T. Levens. *Safely Creating Correct Subclasses without Seeing Superclass Code*. In OOP-SLA '00 Conference Proceedings.

Derek Rayside and Gerard T. Campbell. *An Aristotelian Understanding of Object-Oriented Programming*. OOP-SLA '00 Conference Proceedings.

NOTES

PART III: GRAPHICAL USER INTERFACE PROGRAMMING

CHAPTER 12



Washington Canoe Club

JAVA SWING API OVERVIEW

LEARNING OBJECTIVES

- *Describe the difference between AWT and Swing*
- *State the purpose of the computer screen coordinate system*
- *Utilize windows in your Java programs*
- *State the differences between types of windows*
- *State the difference between a component and a container*
- *Describe the purpose of layout managers and their use*
- *Create various components and organize them into a moderately complex GUI*
- *List and describe packages of the Java Swing API*

INTRODUCTION

This chapter assumes the reader has had no experience with GUI (Graphical User Interface) programming. It describes the computer screen's coordinate system, how to create windows, how to create different graphical components and how to organize them into windows and containers. It does not describe all aspects of Swing programming, as that would require an entire book in itself. Instead, it concentrates on the basics, and attempts to give a solid foundation upon which the reader can build. This chapter is the basis for the next chapter on events and culminates in a moderately complex interface. It is left to the next chapter to add event handling logic, turning this interface into a fully interactive program.

When I am learning a new computer technology, I like to test out concepts early in the learning process by coding and seeing the results. Often, I like to see it “happen” even before I understand it all. Something about seeing it happen gives me added inspiration to learn the details. So, on the assumption that what I like must be what you like, this chapter takes a hands-on approach to learning. We will create several tiny GUI applications as we learn. Certain concepts will be discussed formally only after we have coded and played with them a bit.

We have a lot of ground to cover, so after you have compiled and run each application, I strongly encourage you to consult the javadocs for additional ideas and information. You are also encouraged to change the code experimentally and observe the results. The final portion of this chapter might seem overwhelming unless you have become quite comfortable with the sections that precede it.

AWT AND SWING

In 1996, when Java was first becoming popular, GUI components were part of the “abstract window toolkit” or AWT package. By design, AWT components were dependent on host platform resources for their appearance and behavior. This meant that a Java button on a Macintosh, for instance, would look and behave like a native Macintosh button, but when running on a Microsoft Windows platform, for instance, would look and behave like a native Microsoft Windows button. The API for the Java components was consistent from platform to platform but each component deferred to a platform-specific component “peer” that was responsible for its actual appearance and behavior. So, `java.awt.Button` deferred to `java.awt.peer.ButtonPeer`, `java.awt.List` deferred to `java.awt.peer.ListPeer` and `java.awt.MenuBar` deferred to `java.awt.peer.MenuBarPeer` (*to name just a few*). At first, this was deemed to be a strength of the AWT design. For Java applications to have conquered multiple platforms — even to the point of looking as though they were native to each platform — seemed like the quintessential achievement of the “write once, run anywhere” dream.

Unfortunately, this approach proved to be limiting. “write once, run anywhere” could keep its promise only by supporting just those features that all platforms had in common. If, for example, Macintosh had a great GUI feature that Microsoft Windows didn't, the AWT couldn't provide a platform independent API that included that feature because it would break on Microsoft Windows platforms. Consequently, the feature wouldn't be offered at all. Being constrained to the intersection of platform features virtually guaranteed that Java GUIs would have fewer capabilities than their non-Java counterparts. Furthermore, the supposed strength of being native proved to be a nuisance due to the unavoidable inconsistencies between platforms.

It was inevitable that solutions to this problem would be invented and, in 1997, a project at Sun Microsystems with code name “Swing” was responsible for creating a package of pure Java GUI components that was not dependent on native resources. These classes were known as the JFC or “Java Foundation Classes” and were initially offered as a pure Java extension to the JDK. They became a standard part of the Java 2 Platform with version 1.2. Because they are not dependent on native resources, Swing components are known as “lightweight” components to distinguish them from “heavyweight” AWT components. Being pure Java, they can offer a virtually unlimited number of features — anything that can be expressed in Java. Consequently, the Swing package has grown to be immense and very powerful, overshadowing the AWT in many respects. There are still occasions when the AWT is the best choice (*e.g. when using the Java3D extension package which is designed to take advantage of graphics hardware that can only draw on native resources*), but these are the exceptions. Whenever possible, it is advisable to choose Swing over AWT.

NAMING CONVENTIONS

AWT components are located in the `java.awt` package while Swing components are located in the `javax.swing` package. In general, if a Swing component has an AWT counterpart, its name is the same as its AWT counterpart's name except that it is preceded by the letter "J". In this chapter, when clarity requires, a component will be referred to by its fully qualified name as in "`java.awt.Component`" (*for an AWT component*) or "`javax.swing.JComponent`" (*for a Swing component*). In other cases it will be referred to by its short name as in "`Component`" or "`JComponent`". If a component archetype is being discussed and it doesn't matter whether it's AWT or Swing, then it will be referred to by its AWT name without capitalization as in "`component`" or "`button`". Hopefully this will be clear in context. This chapter concentrates on Swing GUI programming and talks about the AWT only where it has relevance due to inheritance or because there was never a need to supplant it.

THE MATHEMATICS OF GUIs

COORDINATE SYSTEMS

The art of creating a GUI is most certainly the art of illusion. When we press a button in a window with a mouse and see it "depress", when we say that one window is hiding part of another because it is "in front of" the other, when we say that an icon is "transparent" because we can see what is "behind" it, we are willingly succumbing to this illusion. The GUI elements on the screen seem somehow tangible and we can hardly avoid thinking of them as "objects", but it all boils down to nothing more than pixel coordinates and colors painted on a computer screen canvas. Therefore, it is appropriate to discuss the computer screen itself briefly from a GUI designer's perspective.

The computer screen can be thought of as a 2-dimensional Cartesian coordinate system with horizontal x-axis and vertical y-axis. The origin ($x = 0, y = 0$) is the top-left corner of the screen. Distances are measured in pixels with x-axis coordinates increasing rightwards and y-axis coordinates increasing downwards. Yes, downwards! This is at odds with what we all learned in Algebra 101, but it is natural for the computer which paints the screen pixel by pixel the same way we read a book: moving left to right and top to bottom. Take a moment to compare the difference between the standard mathematical coordinate system and the coordinate system of a computer screen. To help illustrate the difference, the line with equation $y = x + 5$ is graphed in both systems in Figures 12-1 and 12-2.

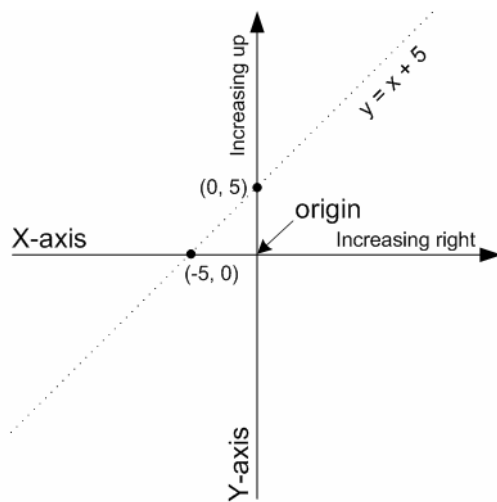


Figure 12-1: Standard Algebraic Coordinate System

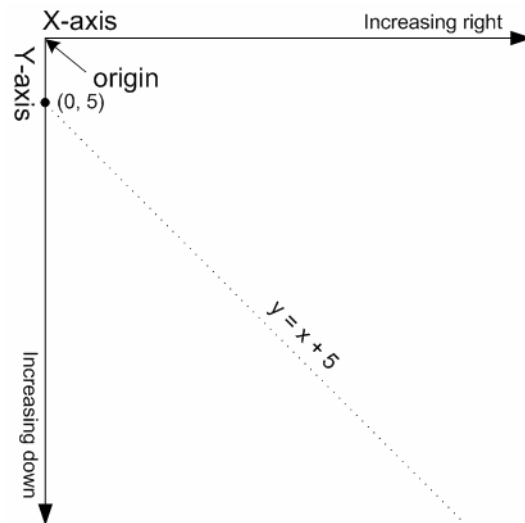


Figure 12-2: Standard Computer-Screen Coordinate System

COMPONENTS AND THEIR BOUNDS

GUI components such as windows and buttons generally occupy rectangular areas, so their exact size and location can be given by the x- and y-coordinates of their top-left corner, their width and their height. Together, these four values define what is known as the *bounds* of a component. Just as the computer screen has a coordinate system, so do all GUI components. Every component considers its own top-left corner to be its local origin with x-coordinates increasing rightward and y-coordinates increasing downward. The bounds of a component are always expressed in terms of its container's coordinate system. Figure 12-3 shows a window with bounds $x = 100$, $y = 275$, $\text{width} = 600$, $\text{height} = 350$. Its bounds are in screen coordinates because the screen contains the window. The “Click Me” button which is inside the window, has bounds $x = 125$, $y = 125$, $\text{width} = 300$, $\text{height} = 100$. Its bounds are in window coordinates because the window contains the button.

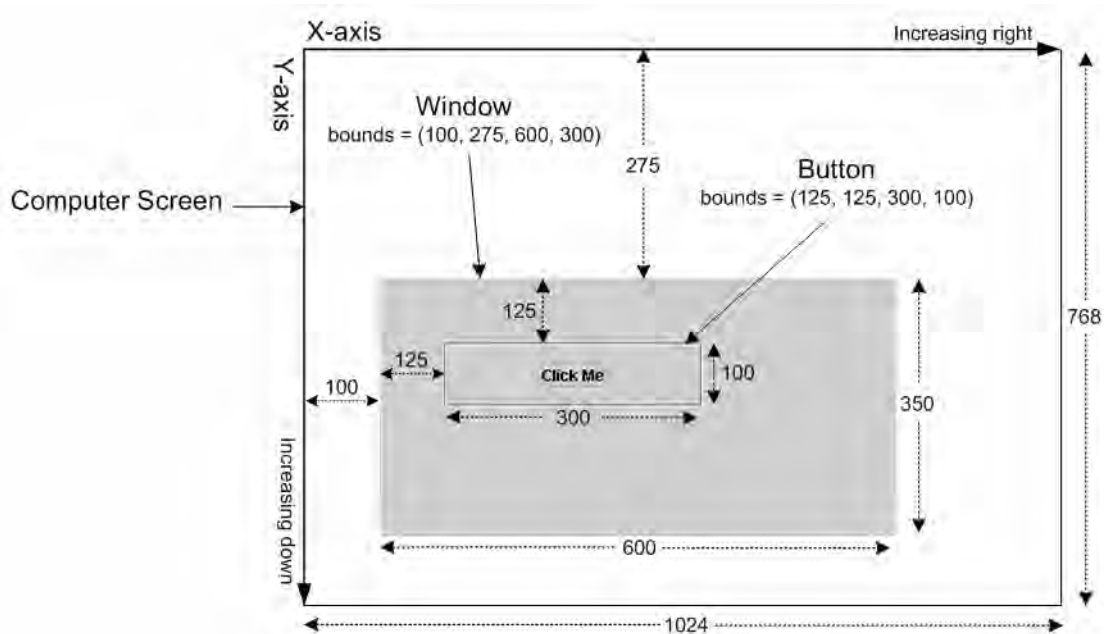


Figure 12-3: Components and Bounds

TOP-LEVEL CONTAINERS

In order for an application to have a GUI, it must have, at minimum, a window visible on the computer screen. Most applications we deal with operate within the confines of a window. Windows look somewhat different from application to application and from platform to platform, but they basically amount to a rectangular section of the screen that is devoted to displaying output from a program and accepting input from the user. In Java, all types of windows are instances or descendants of `java.awt.Window`. A *component* is an instance or descendant of `java.awt.Component` and has a visible representation on the screen that may or may not respond to user input. Buttons, checkboxes and textareas are all familiar examples of components. A *container* is an instance or descendant of `java.awt.Container` and is nothing more than a component that can “contain” other components. Thus, a window is a component because it has a visible representation, and it is a container because it can contain other components. A window is a *top-level* container because it lives directly on the user’s desktop and cannot be contained by any other container. AWT offers three types of windows: `Window`, `Frame` and `Dialog`. Swing offers `JWindow`, `JFrame` and `JDialog` which extend respectively from their AWT counterparts as figure 12-4 illustrates.

```

java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        javax.swing.JWindow
        java.awt.Frame
          javax.swing.JFrame
        java.awt.Dialog
          javax.swing.JDialog

```

Figure 12-4: Top-Level Container Hierarchy

WINDOW AND JWINDOW

A *window* is a top-level container with no borders and no menubar. By itself, it isn't much more than a rectangular section of screen that has been filled with a default background color and reserved for the use of the application. `java.awt.Window` is the root class of all top-level containers. `JWindow` is the Swing version of `Window`. Its looks are deceptively simple. In addition to other components not mentioned here, it contains a container called the *content pane* which is responsible for managing application specific content. When we put content (*buttons, textareas, pictures, etc.*) into a `JWindow`, we actually must add them to this content pane.

Figure 12-5 shows a screen shot of an empty `JWindow`; figure 12-6 shows the structure of the `JWindow`.



Figure 12-5: Screen Shot of an Empty JWindow

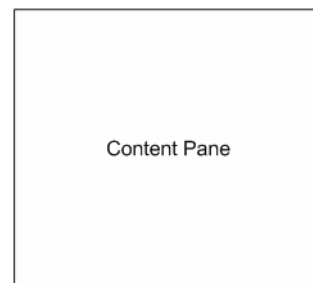


Figure 12-6: Structure of a JWindow

FRAME AND JFrame

A *frame* is usually what we think of when we say “window”. A frame typically includes a close-box, a title bar, a border that can be dragged to resize the frame, and an optional menu bar. `JFrame` is the Swing version of `Frame`. Similarly to `JWindow`, it has a content pane that is responsible for managing application-specific content. `JFrame` is the starting point for most GUI applications because of the window decorations it provides and because it isn't dependent on the existence of a previously created window.

Figures 12-7 and 12-8 show an empty `JFrame` and its corresponding structure. Figures 12-9 and 12-10 show a `JFrame` with a menubar and its corresponding structure.

DIALOG AND JDialog

A *dialog* is a top-level window typically used to take some form of input from the user. Like a frame, it has a title and a border and an optional menubar. A dialog is very similar to a frame except that it is dependent on a parent frame or dialog. When its parent is hidden or closed, the dialog is hidden or closed. Also, a dialog can be *modal* which means that it can block user input to other parts of the application until the dialog is closed. `JDialog` is the Swing version of `Dialog`. `JDialog` has the same structure as a `JFrame`. Figure 12-11 shows a `JDialog` with a label and three buttons.

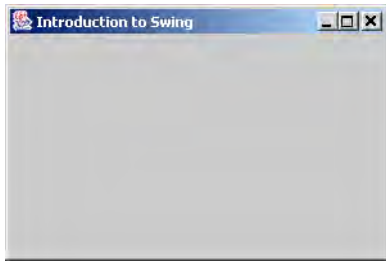


Figure 12-7: Screenshot of an Empty JFrame

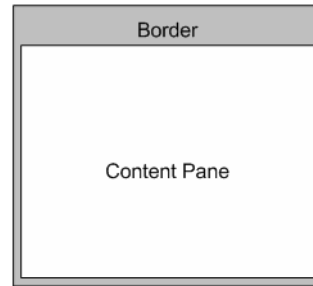


Figure 12-8: Structure of an Empty JFrame

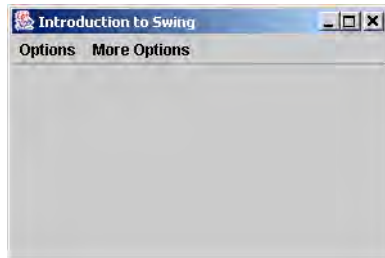


Figure 12-9: JFrame with Menubar

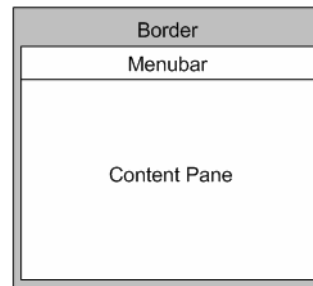


Figure 12-10: Structure of JFrame with Menubar



Figure 12-11: A JDialog with a Label and Three Buttons

OUR FIRST GUI PROGRAM

Let's have some fun and write a GUI application. Our first GUI program will be "minimal" and "well-behaved". By "minimal" I mean that it will display the simplest possible interface with which the user can interact. By "well-behaved" I mean that it will respond to user input in an appropriate manner. Compile and run the following code.

12.1 chap12.TestFrame.java

```

1     package chap12;
2     import javax.swing.JFrame;
3
4     public class TestFrame {
5         public static void main(String[] arg) {
6             JFrame frame = new JFrame("Introduction to Swing");
7             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8             frame.setBounds(200, 100, 300, 200);
9             frame.setVisible(true);
10        }
11    }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap12/TestFrame.java
java -cp classes chap12.TestFrame

```

Figure 12-12 shows the results of running this example. Check out the resulting application. Wow! We've written practically no code at all but look at all that has been provided to us. Click on the window decorations: the close box,

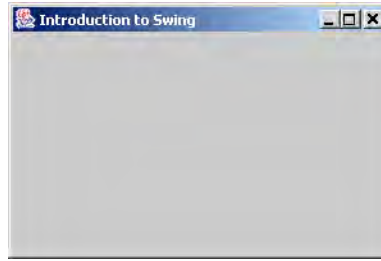


Figure 12-12: TestFrame GUI

and minimize and maximize buttons. Resize the window. This window has the same behavior in all respects as the standard window on your computer no matter the operating system because it *is* a standard operating system window.

Explanation of the Code

On line 6 we used a `JFrame` because it provides us with the operating system's standard decorated window, and because it fully supports Swing's component architecture. We passed a `String` parameter into the constructor so the `JFrame` could display a title in its title bar. A well-behaved program usually displays titles on the windows it creates to distinguish them from windows created by other applications.

Since our program only creates one window, quitting the application when the user attempts to close the window is the expected behavior. For this reason we called `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` on line 7. The available options for the default behavior are the following:

- `EXIT_ON_CLOSE` - quits the application;
- `DO_NOTHING_ON_CLOSE` - does nothing;
- `HIDE_ON_CLOSE` - makes the window invisible but keeps it in memory for redisplay;
- `DISPOSE_ON_CLOSE` - destroys the window and frees up associated memory.

You can take some time here to experiment with the code. Change line 7 to use `DO_NOTHING_ON_CLOSE`, compile and run the program. Now change it to use `HIDE_ON_CLOSE`. Do you notice the difference in behavior between `HIDE_ON_CLOSE` and `EXIT_ON_CLOSE`?

On line 8 our well-behaved program sets the bounds of the window rather than leaving it to platform defaults. `JFrame` inherits the `setBounds()` method from `java.awt.Component` (see table 12-1). The parameters to the `setBounds()` method are in pixels and are interpreted in the coordinate system of the component's container, which is in this case the computer screen. The width and height parameters include the frame's border decorations, so the content pane is somewhat smaller than its total bounds. Comment out line 8, compile and run again to see the results. Change the bounds parameters, compile and run. What happens if you give a negative value for one of the parameters, for instance? What happens if you set the width and height to zero?

Method Name & Purpose
public void setBounds(int x, int y, int width, int height) Sets the location and size of the component.
public void setBounds(Rectangle r) Sets the location and size of the component to match the specified <code>Rectangle</code> .

Table 12-1: `java.awt.Component.setBounds()` Methods

On line 9 we called `JFrame.setVisible(true)` so that the window would be visible. Simple enough! Comment out this line, compile and run the program again to see (*or not see*) the results.

Top-Level Containers API

OK, that's enough fun for now. It's time to look at the API for the top-level containers. While it is generally my intention to encourage you to consult the javadocs, I couldn't in good conscience send you there on your own right now without some help. If you look at the javadocs for `java.awt.Window`, for example, you will find a mind-numbing assortment of methods, many of which have no relevance to a beginner. Indeed, some of them are so technical that you might never need them, but they're there just in case. If I counted correctly, `Window` defines or overrides 51 different methods. But remember that `Window` extends from `Container`, which extends from `Component`, which extends from `Object`. From `Container` it inherits 59 methods, from `Component` it inherits 161, and from `Object` it inherits 9. Adding these numbers together we get $51 + 59 + 161 + 9 = 280$ methods available to `Window`! This number is slightly inflated because it double-counts overridden methods, but it does highlight the enormity of the AWT/Swing API. Unfortunately, the javadocs don't organize methods by programmer level (*novice/expert*), area of functionality, or even relative importance. They arrange them alphabetically. In this chapter, we will only cover only 30 of the top-level container methods, but that will be enough to get you pretty far. After the next chapter adds a few event handling methods, you will have possibly all you'll ever need to create a great variety of complex GUI applications.

Top-Level Container Constructors

There are many constructors for top-level containers depending on which type you are creating. Most of them are listed in table 12-2. Take a look at them now.

Top-Level Containers Constructor Chart						
Window types: W = Window, F = Frame, D = Dialog, JW = JWindow, JF = JFrame, JD = JDialog						
Constructor Parameters	W	F	D	JW	JF	JD
() [no parameters] Frame and JFrame: Constructs an initially invisible frame. JWindow: Constructs an initially invisible window. JDialog: Constructs an initially invisible non-modal dialog.		✓		✓	✓	✓
(Window owner) Constructs a new invisible window with the specified Window as its owner.	✓			✓		
(Frame owner) Constructs a new invisible window or dialog with the specified Frame as its owner.	✓		✓	✓		✓
(Dialog owner) Constructs an initially invisible, non-modal dialog without a title and with the specified owner dialog.			✓			✓
(String title) Constructs a new, initially invisible frame with the specified title.		✓			✓	
(Frame owner, String title) Constructs an initially invisible, non-modal dialog with the specified owner frame and title.			✓			✓
(Dialog owner, String title) Constructs an initially invisible, non-modal dialog with the specified owner dialog and title.			✓			✓
(Frame owner, boolean modal) Constructs an initially invisible dialog without a title and with the specified owner frame and modality.			✓			✓

Table 12-2: Top-Level Containers Constructor Chart

(Dialog owner, boolean modal) Constructs an initially invisible dialog without a title and with the specified owner dialog and modality.						✓
(Frame owner, String title, boolean modal) Constructs an initially invisible dialog with the specified owner frame, title, and modality.		✓				✓

Table 12-2: Top-Level Containers Constructor Chart

From this table we can make several observations:

- **Observation 1: Window, JWindow, Dialog and JDialog offer constructors that take owner parameters.** Windows that are owned by other windows are dependent on their owner windows. They are minimized when their owner is minimized, they are closed when their owner is closed, and they are hidden or shown when their owner is hidden or shown.
- **Observation 2: Frame and JFrame don't offer constructors that take an owner parameter.** Actually, they are the only top-level containers whose construction doesn't require the existence of a previously constructed top-level container.

Dialogue with a Skeptic

Skeptic: Wait a minute! I'm looking right at the javadocs and I see that JWindow and JDialog offer constructors that don't require an owner parameter. Doesn't that mean that their construction doesn't require the existence of a previously constructed top-level container either?

Author: Yes, the javadocs do seem to suggest that, but it's not true. If you don't pass an owner into a JWindow or JDialog constructor, or if you pass an owner that is null, Swing assigns them an owner Frame courtesy of `javax.swing.SwingUtilities.getSharedOwnerFrame()`.

Skeptic: [Quickly looks up *SwingUtilities* in the javadocs. Not finding the method *getSharedOwnerFrame*, he returns a bit suspicious.] What method did you say? I didn't find it in the javadocs.

Author: [Good-natured chuckle] Oh, it's not in the javadocs because it's package private, but you can find the actual source code in the "src.zip" file which should be located right inside your Java installation directory. Look at the constructors in `javax.swing.JWindow.java` and you will see references to `SwingUtilities.getSharedOwnerFrame()`. Then if you look at the source for `javax.swing.SwingUtilities.java` —

Skeptic: Wait a minute while I find it!

Author: I'll wait.

Skeptic: [After navigating his computer's file system, he finds the Java installation directory and opens the "src.zip" file] Ok, I see the "SwingUtilities.java" file. [Opening up "SwingUtilities.java" in an editor and scanning the file] Now I've found the `getSharedOwnerFrame` method! [Reading] "static Frame `getSharedOwnerFrame()` throws `HeadlessException`".

Author: Great! Just look at the comments for now. What do they say?

Skeptic: [Reading] "Returns a toolkit-private, shared, invisible Frame to be the owner for *JDialogs* and *JWindows* created with null owners."

Author: Very good! For now we'll continue with our observations. But feel free to browse any of the source code on your own later. It can often be very helpful.

- **Observation 3: Only Frame, JFrame, Dialog and JDialog constructors offer a title parameter.** This makes sense since they're the only ones with title bars.
- **Observation 4: Only Dialog and JDialog offer the modal parameter.** If the modal parameter is true, the dialog will block user input to the owner window until the dialog has been closed. If it's false, input to the owner window will not be blocked.
- **Observation 5: All constructors create initially invisible windows.** You must call `setVisible(true)` or `show()` to make the window visible.

TOP-LEVEL CONTAINER METHODS

Read through tables 12-3 through 12-7 to familiarize yourself with many of the top-level container methods. Note that the methods in these tables do not include the many methods inherited from Container from which, as you may remember, all window types extend.

Method Name and Purpose
public void show() Makes the window visible.
public void hide() Hide this window, its subcomponents, and all of its owned children.
public boolean isShowing() Checks if this window is showing on the screen.
public void toFront() If this window is visible, brings this window to the front and may make it the focused window.
public void toBack() If this window is visible, sends this window to the back and may cause it to lose focus or activation if it is the focused or active window.
public boolean isFocused() Returns whether this window is focused.
public boolean isActive() Returns whether this window is active. Only frames and dialogs may be active. The active window is always either the focused window, or the first frame or dialog that is an owner of the focused window.
public void setLocationRelativeTo(Component) Sets the location of the window relative to the specified component.
public void pack() Causes this window to be sized to fit the preferred size and layouts of its subcomponents.
public String getWarningString() Gets the warning string that is displayed with this window. You will see a warning string if the window is considered insecure. This may happen, for instance, if the window is created by an Applet running in a browser.
public Window getOwner() Returns the owner of this window.
public Window[] getOwnedWindows() Return an array containing all the windows this window currently owns.
public void dispose() Releases all of the native screen resources used by this window, its subcomponents, and all of its owned children.

Table 12-3: Methods Available to All Descendants of Window

Method Name and Purpose
public String getTitle() Gets the title of the frame or dialog.
public void setTitle(String title) Sets the title for this frame or dialog to the specified string.

Table 12-4: Methods Available to Frame, JFrame, Dialog, JDialog Only

public boolean isResizable() Indicates whether this frame or dialog is resizable by the user.
public void setResizable(boolean resizable) Sets whether this frame or dialog is resizable by the user.
public boolean isUndecorated() Indicates whether this frame or dialog is undecorated.
public void setUndecorated(boolean undecorated) Disables or enables decorations for this frame or dialog.

Table 12-4: Methods Available to Frame, JFrame, Dialog, JDialog Only

Method Name and Purpose
public static Frame[] getFrames() Returns an array containing all Frames created by the application.
public int getExtendedState() Gets the state of this frame represented as a bitwise mask. Any of the following may be bitwise or'd together: NORMAL, ICONIFIED, MAXIMIZED_HORIZ, MAXIMIZED_VERT and MAXIMIZED_BOTH.
public void setExtendedState(int state) Sets the state of this frame represented as a bitwise mask. (See getExtendedState() above).
public Image getIconImage() Gets the image to be displayed in the minimized icon for this frame. This is also the image displayed in the top-left corner of the title bar.
public void setIconImage(Image image) Sets the image to be displayed in the minimized icon for this frame. (See getIconImage() above).

Table 12-5: Methods Available to Frame, JFrame Only

Method Name and Purpose
public Container getContentPane() Returns the Container which is the content pane for this window. Components added to any Swing top-level container must be added to the container's content pane. Although, as descendants of Container they inherit all of the add methods, Swing top-level containers have overridden these methods to throw a runtime exception. In other words, your code would compile but not run. So remember: when using JWindow, JFrame and JDialog add components to the content pane!
public void setContentPane(Container contentPane) Sets the content pane property for this window.

Table 12-6: Methods Available to JWindow, JFrame, JDialog Only

Method Name and Purpose
public JMenuBar getJMenuBar() Returns the menubar set on this frame or dialog.
public void setJMenuBar(JMenuBar menu) Sets the menubar for this frame or dialog.

Table 12-7: Methods Available to JFrame, JDialog Only

public int getDefaultCloseOperation()

Returns the operation which occurs when the user initiates a "close" on this frame or dialog. This will either be DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE or EXIT_ON_CLOSE.

public void setDefaultCloseOperation(int operation)

Sets the operation which will happen by default when the user initiates a "close" on this frame or dialog. (See getDefaultCloseOperation() above).

Table 12-7: Methods Available to JFrame, JDialog Only

Quick Review

A *component* is any GUI object that has a visible representation on the screen. A *container* is a component that can "contain" other components. A *top-level* container is a container that exists on the computer screen and can't be contained by another container. Java GUI applications must start with a top-level container otherwise known as a window. There are three types of windows: Window, Frame, Dialog. Their Swing counterparts are JWindow, JFrame and JDialog. All Swing top-level containers have a content pane to which GUI components must be added. JFrame is comprised of a title bar, border, optional menubar and a content pane. Most Swing applications will begin with a JFrame since it isn't dependent on the existence of a previously created window.

ORGANIZING COMPONENTS INTO CONTAINERS

Before we discuss the components themselves in detail, we will discuss how to organize them into containers. From this standpoint, all components act alike. To not distract from the subject at hand, the next few programs will only use the three familiar component archetypes: the button, the text area and the label which are implemented by JButton, JTextArea and JLabel respectively. The code should be self-explanatory, but feel free to consult the javadocs for additional information. At its basic level, creating a GUI is nothing more than putting components in containers. Let's begin doing just that.

Absolute Positioning

The following code augments example 12.1 by adding a button, text area and label to the JFrame. First we call `ContentPane.setLayout(null)` in line 17 to instruct the content pane to put components where we explicitly say. The significance of that line will be more apparent when we discuss `LayoutManagers`. Then, for each component we add, we use the `setBounds()` method to set the exact size and location of each component (*as in lines 20, 25 and 31*). This is called absolute positioning. Since each component is added to the content pane of the JFrame, the parameters passed into the `setBounds()` method are interpreted in the coordinate system of the content pane – not the JFrame itself.

12.2 chap12.TestFrameWithContents.java

```

1      package chap12;
2      import java.awt.Container;
3
4      import javax.swing.JButton;
5      import javax.swing.JFrame;
6      import javax.swing.JLabel;
7      import javax.swing.JTextArea;
8
9      import utils.TreePrinterUtils;
10
11     public class TestFrameWithContents {
12         public static void main(String[] arg) {
13             JFrame frame = new JFrame("TestFrameWithContents");
14             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15             frame.setBounds(200, 200, 200, 200);
16             Container contentPane = frame.getContentPane();
17             contentPane.setLayout(null);
18
19             JButton button = new JButton("I'm a JButton");
20             button.setBounds(0, 0, 200, 40);

```

```

21     contentPane.add(button);
22
23     JTextArea textArea =
24     new JTextArea("I'm a JTextArea and am designed to hold a lot of text.");
25     textArea.setBounds(0, 40, 200, 100);
26     textArea.setLineWrap(true);
27     textArea.setWrapStyleWord(true);
28     contentPane.add(textArea);
29
30     JLabel label = new JLabel("I'm a JLabel");
31     label.setBounds(0, 140, 200, 40);
32     contentPane.add(label);
33
34     frame.setVisible(true);
35     TreePrinterUtils.printLayout(contentPane);
36 }
37 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap12/TestFrameWithContents.java
java -cp classes chap12.TestFrameWithContents

```

Figure 12-13 shows the results of running example 12.2. Figure 12-14 shows component layout information that has been printed to the console.

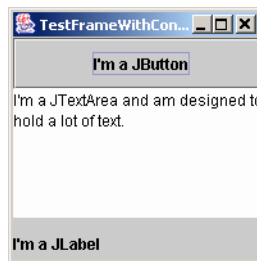


Figure 12-13: TestFrameWithContents GUI

```

JPanel [0, 0, 192, 172] (null: 3 children)
+--JButton [0, 0, 200, 40]
+--JTextArea [0, 40, 200, 100]
+--JLabel [0, 140, 200, 40]

```

Figure 12-14: TestFrameWithContents Console Output

Notice the console output resulting from `TreePrinterUtils.printLayout(contentPane)`. `TreePrinterUtils.printLayout` is a handy tool for displaying the layout of a container in textual form. It will be utilized by all the remaining programs in this chapter to help clarify interface layouts. It progresses recursively through a component tree displaying the containment hierarchy and printing information about every component it encounters. For each component, it prints the component's type followed by its bounds in square brackets. Then, if the component is a container and contains at least one child component, it prints the layout manager type and the number of child components inside parentheses. Layout managers will be discussed shortly.

By setting the bounds of each component explicitly, we have complete control of how we want our content pane to look. Great! With complete control, however, comes complete responsibility. Resize the window larger and smaller and notice the results. See Figures 12-15 and 12-16 to see the results of resizing the window larger and smaller.

Oops! Wouldn't it be nicer if the components were automatically resized when we resized the window? Well, with a lot more coding we could write methods into our application that notice when the window is resized and then recalculate the bounds of the components so that they look good at the new window size. We *could* do this but we won't. AWT provides us with a very handy mechanism for doing this and more. It is called the *layout manager*.

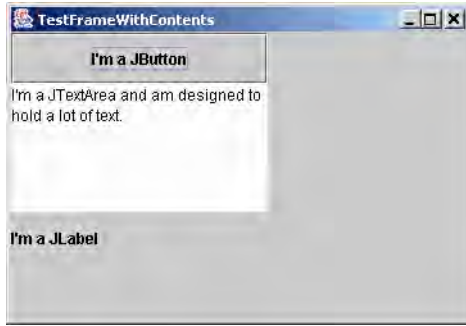


Figure 12-15: TestFrameWithContents Resized Larger

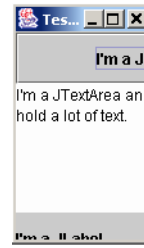


Figure 12-16: TestFrameWithContents Resized Smaller

LAYOUT MANAGERS

`java.awt.LayoutManager` is an interface that declares methods for laying out components in a container. A layout manager earns its keep every time something happens in a container that might necessitate the repositioning of that container's components. That includes resizing of the container, the addition or removal of children components and other events. While all components can make suggestions regarding what size to draw them (*see the Component methods `getPreferredSize()`, `getMinimumSize()` and `getMaximumSize()`*) the layout manager is the final authority for actually setting the bounds of all components by calling `Component.setBounds()`. By installing a predefined or custom layout manager into a container through `Container.setLayout()` we can avoid much of the headache of managing a container's layout ourselves.

FlowLayout

`FlowLayout` arranges components similarly to the way a word-processor arranges text in a document. By default, it fills the container from left to right and top to bottom, “wrapping” to the next “line” components that would otherwise spill over the right edge of the container. By default, components in each line are centered and are separated from each other by a horizontal and vertical gap of 5 pixels. Alignment options and gap sizes may be passed into the constructor or set after the fact through `FlowLayout.setHgap(int)`, `FlowLayout.setVgap(int)` and `FlowLayout.setAlignment(int)`. Options for alignment include:

- `FlowLayout.CENTER` (the default) - centers each row of components.
- `FlowLayout.LEFT` - left-justifies each row of components.
- `FlowLayout.RIGHT` - right-justifies each row of components.
- `FlowLayout.LEADING` - justifies each row of components to its container's leading edge.
- `FlowLayout.TRAILING` - justifies each row of components to its container's trailing edge.

`FlowLayout` always sizes components to their preferred size regardless how big or small the container actually is. Example 12.3 modifies example 12.2 by using `FlowLayout` as the layout manager for the content pane in line 18.

12.3 chap12.TestFrameWithFlowLayout.java

```

1     package chap12;
2     import java.awt.Container;
3     import java.awt.FlowLayout;
4
5     import javax.swing.JButton;
6     import javax.swing.JFrame;
7     import javax.swing.JLabel;
8     import javax.swing.JTextArea;
9
10    import utils.TreePrinterUtils;
11
12    public class TestFrameWithFlowLayout {
13        public static void main(String[] arg) {
14            JFrame frame = new JFrame("TestFrameWithFlowLayout");
15            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16            frame.setBounds(200, 200, 200, 200);
17            Container contentPane = frame.getContentPane();
18            contentPane.setLayout(new FlowLayout());
19        }
    }

```

```

20     JButton button = new JButton("I'm a JButton");
21     button.setBounds(0, 0, 200, 40);
22     contentPane.add(button);
23
24     JTextArea textArea =
25         new JTextArea("I'm a JTextArea and am designed to hold a lot of text.");
26     textArea.setBounds(0, 40, 200, 100);
27     textArea.setLineWrap(true);
28     textArea.setWrapStyleWord(true);
29     contentPane.add(textArea);
30
31     JLabel label = new JLabel("I'm a JLabel");
32     label.setBounds(0, 140, 200, 40);
33     contentPane.add(label);
34
35     frame.setVisible(true);
36     TreePrinterUtils.printLayout(contentPane);
37 }
38 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap12/TestFrameWithFlowLayout.java
java -cp classes chap12.TestFrameWithFlowLayout

```

Figure 12-17 shows the results of running example 12.3. Figure 12-18 shows component layout information that has been printed to the console.

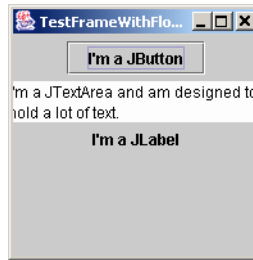


Figure 12-17: TestFrameWithFlowLayout GUI

```

JPanel [0, 0, 192, 172] (FlowLayout: 3 children)
+--JButton [42, 5, 108, 26]
+--JTextArea [-4, 36, 200, 32]
+--JLabel [62, 73, 68, 16]

```

Figure 12-18: TestFrameWithFlowLayout Console Output

Notice that the component bounds in the output of `TreePrinterUtils.printLayout` are different than the bounds to which we set them. Well, what did you expect? After all, it's the layout manager's job to set the bounds – not ours. A layout manager determines the size and location of each component based on its own logic. Although not required, it is certainly reasonable that a layout manager would ignore a component's current bounds when calculating that component's new bounds..

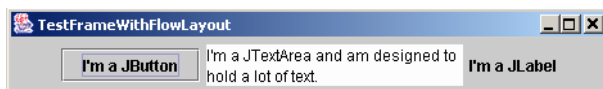


Figure 12-19: TestFrameWithFlowLayout Resized Wider

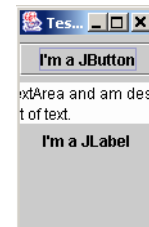


Figure 12-20: TestFrameWithFlowLayout Resized Taller

Figures 12-19 and 12-20 show the effects of resizing the window wider and taller. Notice the wasted space and especially the poor layout of the JTextArea in figure 12-20 whose left and right sides are lopped off by the window's own edges. This is clearly not the best layout manager for this application but it is certainly an improvement over no layout manager at all.

GridLayout

GridLayout divides a container's area into a grid of equally sized cells. The number of cells in each row and column may be determined in the constructor or after construction via `setRows(int)` and `setColumns(int)`. If either the rows or columns are set to zero, this is interpreted as "however many are necessary to fit all the children components". For instance, if a GridLayout containing eight children has its rows set to zero and its columns set to two, then it will create four rows and two columns to contain the eight components. Figure 12-21 shows the grid coordinates and layout for a component using a GridLayout with four rows and two columns. .

0, 0	0, 1
1, 0	1, 1
2, 0	2, 1
3, 0	3, 1

Figure 12-21: Coordinates for a Sample GridLayout with 4 Rows and 2 Columns

Like FlowLayout, GridLayout has configurable horizontal and vertical gap parameters. Unlike FlowLayout, GridLayout ignores components' preferred sizes and guarantees that all components will fit in the container no matter how large or small. In example 12.4 we use GridLayout to see its effects on our tiny application. We also removed the superfluous calls to `Component.setBounds()`.

12.4 chap12.TestFrameWithGridLayout.java

```

1      package chap12;
2      import java.awt.Container;
3      import java.awt.GridLayout;
4      import javax.swing.JButton;
5      import javax.swing.JFrame;
6      import javax.swing.JLabel;
7      import javax.swing.JTextArea;
8      import utils.TreePrinterUtils;
9
10     public class TestFrameWithGridLayout {
11         public static void main(String[] arg) {
12             JFrame frame = new JFrame("TestFrameWithGridLayout");
13             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14             frame.setBounds(200, 200, 200, 200);
15             Container contentPane = frame.getContentPane();
16             contentPane.setLayout(new GridLayout(3, 1));
17
18             JButton button = new JButton("I'm a JButton");
19             contentPane.add(button);
20
21             JTextArea textArea =
22                 new JTextArea("I'm a JTextArea and am designed to hold a lot of text.");
23             textArea.setLineWrap(true);
24             textArea.setWrapStyleWord(true);
25             contentPane.add(textArea);
26
27             JLabel label = new JLabel("I'm a JLabel");
28             contentPane.add(label);
29
30             frame.setVisible(true);
31             TreePrinterUtils.printLayout(contentPane);
32         }
33     }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap12/TestFrameWithGridLayout.java
```

```
java -cp classes chap12.TestFrameWithGridLayout
```

Figure 12-22 shows the results of running example 12.4. Figure 12-23 shows the GUI's component location information printed to the console. Figures 12-24 and 12-25 show the effects of resizing the window wider and taller.



Figure 12-22: TestFrameWithGridLayout GUI

```
JPanel [0, 0, 192, 172] (GridLayout: 3 children)
+--JButton [0, 0, 192, 57]
+--JTextArea [0, 57, 192, 114]
+--JLabel [0, 114, 192, 172]
```

Figure 12-23: TestFrameWithGridLayout Console Output



Figure 12-24: TestFrameWithGridLayout Resized Wider



Figure 12-25: TestFrameWithGridLayout Resized Taller

Well, this is certainly better. No matter how the window is resized, components seem to be resized intelligently. But wait! Do we really need the button and label to occupy the same amount of space as the text area? We could end up with unnecessarily large buttons and labels if the window were really large. Wouldn't it be better if the text area benefited from the extra space while the button and label just displayed at whatever size was best for them?

BorderLayout

FlowLayout and GridLayout layout components based partly on the order in which the components are added to the container. While these two layout managers can be quite useful, they are not as powerful as another set of layout managers that implement the `LayoutManager2` interface. `LayoutManager2` extends `LayoutManager` by adding methods that allow a component to be associated with a *constraints* object when it is added to a container. A constraints object amounts to a set of suggestions as to how a component should be sized and positioned in the container. `LayoutManager2` implementations should do the best possible job arranging components even when faced with varying and perhaps mutually conflicting constraints.

`BorderLayout` is arguably the simplest implementation of `LayoutManager2`. It's ideal for instances when only one particular component should expand or contract to fill space as the container is resized. It divides the container into 5 sections (*`BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST`, and `BorderLayout.CENTER`*) and can manage up to 5 components. The `CENTER` component fills as much space as possible while the other components resize as necessary to display properly and fill in any gaps as is illustrated in figure 12-26.

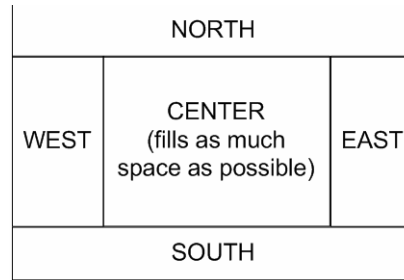


Figure 12-26: BorderLayout Positions

Applying BorderLayout to our tiny application, example 12.5 puts the button in the NORTH section, the label in the SOUTH and the textarea in the CENTER.

12.5 chap12.TestFrameWithBorderLayout.java

```

1      package chap12;
2      import java.awt.BorderLayout;
3      import java.awt.Container;
4
5      import javax.swing.JButton;
6      import javax.swing.JFrame;
7      import javax.swing.JLabel;
8      import javax.swing.JTextArea;
9
10     import utils.TreePrinterUtils;
11
12     public class TestFrameWithBorderLayout {
13         public static void main(String[] arg) {
14             JFrame frame = new JFrame("TestFrameWithBorderLayout");
15             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16             frame.setBounds(200, 200, 200, 200);
17             Container contentPane = frame.getContentPane();
18             contentPane.setLayout(new BorderLayout());
19
20             JButton button = new JButton("I'm a JButton");
21             contentPane.add(button, BorderLayout.NORTH);
22
23             JTextArea textArea =
24                 new JTextArea("I'm a JTextArea and am designed to hold a lot of text.");
25             textArea.setLineWrap(true);
26             textArea.setWrapStyleWord(true);
27             contentPane.add(textArea, BorderLayout.CENTER);
28
29             JLabel label = new JLabel("I'm a JLabel");
30             contentPane.add(label, BorderLayout.SOUTH);
31
32             frame.setVisible(true);
33             TreePrinterUtils.printLayout(contentPane);
34         }
35     }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap12/TestFrameWithBorderLayout.java
java -cp classes chap12.TestFrameWithBorderLayout

```

Figure 12-27 shows the results of running example 12.5. Figure 12-28 shows the GUI component's location information printed to the console. Figures 12-29 and 12-30 show the effects of resizing the window.

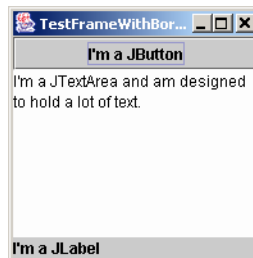


Figure 12-27: TestFrameWithBorderLayout GUI

```
JPanel [0, 0, 192, 172] (BorderLayout: 3 children)
+--JButton [0, 0, 192, 26]
+--JTextArea [0, 26, 192, 130]
+--JLabel [0, 156, 192, 16]
```

Figure 12-28: TestFrameWithBorderLayout Console Output



Figure 12-29: TestFrameWithBorderLayout Resized Wider

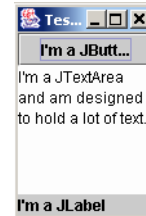


Figure 12-30: TestFrameWithBorderLayout Resized Taller

This is probably the best layout manager for our little application because it resized the text area the way we would expect, leaving the button and label alone except for stretching them horizontally as needed.

GridBagLayout

GridBagLayout is another implementation of `LayoutManager2`. It is quite powerful and quite complex. It can be thought of as `GridLayout` on steroids. Similarly to `GridLayout`, `GridBagLayout` arranges components in a grid. But that's where the similarity stops. Whereas `GridLayout` forces all components to be the same size, `GridBagLayout` allows them to span multiple rows and columns, it allows rows and columns to have varying heights and widths, and it even provides individual sizing and positioning options within each component's grid boundaries. Its only constructor takes no parameters because all other parameters are passed in through a `GridBagConstraints` object that is associated with a component when it is added. We will discuss the `GridBagConstraints` object in detail as we walk through and tweak the following program whose interface was inspired by the controls on my stereo tuner.

12.6 chap12.GridBagLayoutExample.java

```
1 package chap12;
2 import java.awt.Container;
3 import java.awt.GridBagConstraints;
4 import java.awt.GridBagLayout;
5
6 import javax.swing.JButton;
7 import javax.swing.JFrame;
8
9 import utils.TreePrinterUtils;
10
11 public class GridBagLayoutExample {
12     public static void main(String[] arg) {
13         JFrame frame = new JFrame("GridBagLayoutExample");
14         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15
16         Container contentPane = frame.getContentPane();
17         contentPane.setLayout(new GridBagLayout());
18
19         GridBagConstraints gbc = new GridBagConstraints();
20         gbc.fill = gbc.BOTH;
21         gbc.gridwidth = 1;
22         gbc.gridx = 0;
23         gbc.gridy = 0;
24         contentPane.add(new JButton("FM"), gbc);
25         gbc.gridx = 1;
26         contentPane.add(new JButton("AM"), gbc);
27
28         gbc.gridx = 0;
29         gbc.gridy = 1;
30         gbc.gridwidth = 2;
31         contentPane.add(new JButton("Seek - Stereo"), gbc);
32
33         gbc.gridx = 2;
```

```

34         gbc.gridy = 0;
35         gbc.gridwidth = 1;
36         gbc.gridheight = 2;
37         contentPane.add(new JButton("Tuning"), gbc);
38         frame.pack();
39         frame.setVisible(true);
40
41         TreePrinterUtils.printLayout(contentPane);
42     }
43 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap12/GridBagLayoutExample.java
java -cp classes chap12.GridBagLayoutExample

```

Figure 12-31 shows the results of running example 12.6. Figure 12-32 shows the GUI components' location information printed to the console.

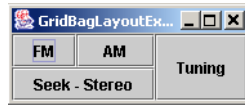


Figure 12-31: GridBagLayoutExample GUI

```

JPanel [0, 0, 183, 52] (GridBagLayout: 4 children)
+--JButton [0, 0, 50, 26]
+--JButton [50, 0, 61, 26]
+--JButton [0, 26, 111, 26]
+--JButton [111, 0, 72, 52]

```

Figure 12-32: GridBagLayoutExample Console Output

GridBagConstraints

A GridBagConstraints object is associated with each component added to a container with a GridBagLayout layout manager. We will discuss here the various fields of the GridBagConstraints object.

gridx, gridy

The *gridx* and *gridy* fields are integers that determine the column and row of the top left corner of the component. Row and column numbers start at zero. The special value GridBagConstraints.RELATIVE instructs the GridBagLayout to place the component next to the component added previously.

gridwidth, gridheight

The *gridwidth* and *gridheight* fields are integers that determine how many columns and rows the component should span. In example 12.6, the “FM” button is placed at *gridx* = 0, *gridy* = 0 and is given *gridwidth* = 1, *gridheight* = 1 so that it occupies a single grid cell at the top left corner of the window. The “AM” button has the same *gridwidth* and *gridheight* but is moved rightward to the next column by setting *gridx* = 1. The “Seek-Stereo” button is located at *gridx* = 0, *gridy* = 1 and is given *gridwidth* = 2 so that it will stretch horizontally across two columns. The “Tuning” button was placed at *gridx* = 2, *gridy* = 0 and given *gridheight* = 2 so that it would stretch vertically across two rows.

fill

The *fill* field is an integer that determines how a component should fill its space if the space allotted to it is larger than its preferred size. Options for fill include:

- HORIZONTAL - stretches the component horizontally.
- VERTICAL - stretches the component vertically.
- BOTH - stretches the component horizontally and vertically.
- NONE - means don't stretch the component.

Figure 12-33 shows the results of setting fill to HORIZONTAL in the constraints for the “Tuning” button.



Figure 12-33: GridBagLayoutExample GUI Variation 1

ipadx, ipady

The *ipadx* and *ipady* fields are integers that determine the minimum padding to be added to the component's width and height. (*Pneumatic: ipad = internal padding*) Note: the API states that the width and height of the component are increased by $ipadx*2$ and $ipady*2$. This however is not the behavior of Java 1.4 where the width and height are only increased by *ipadx* and *ipady*. Figure 12-34 shows the results of setting *ipadx* and *ipady* to 100 in the constraints for the “Tuning” button.

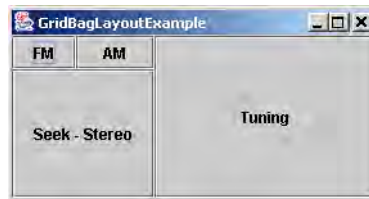


Figure 12-34: GridBagLayoutExample GUI Variation 2

INSETS

The *insets* field is an instance of `java.awt.Insets` and determines how much space there should be on all sides of the component. The insets object has four fields: left, right, top and bottom. Whereas *ipadx* and *ipady* increase the size of the component, insets increases the minimum display area for the component without affecting the size of the component itself. Figure 12-35 shows the results of setting insets to new `Insets(50, 50, 50, 50)` in the constraints for the Tuning button. Notice that the display area for the Tuning button is the same as in the previous GUI but the extra space was added to the outside of the button rather than inside the button.

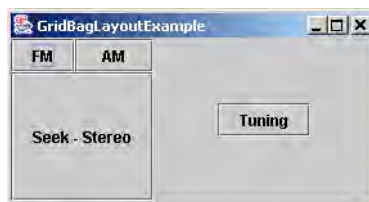


Figure 12-35: GridBagLayoutExample GUI Variation 3

weightx, weighty

The *weightx* and *weighty* fields are floating values ranging from 0.0 to 1.0. If, after all other calculations have been made, the `GridBagLayout` has calculated the combined area of all the components to be smaller than the container's available area, `GridBagLayout` will distribute the extra space to each row and column according to their respective *weightx* and *weighty* values. Rows and columns with higher weights will be given more extra space than

rows and columns with lower weights. A value of 0.0 tells it to not add any extra space. Figure 12-36 shows the results of setting `weightx` and `weighty` to 1 in the constraints for the Tuning button. You will have to drag the window to a larger size for the effects to be manifested.

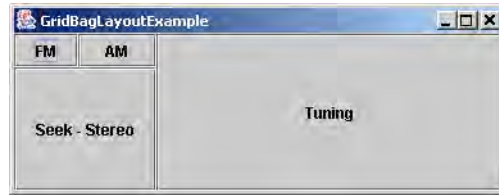


Figure 12-36: GridBagLayoutExample GUI Variation 4

ANCHOR

The *anchor* field is an integer that determines where, within its space, a component is positioned, and has effect only when the component is smaller than the space allotted to it. This can only occur when `fill` is not `BOTH`. Options for `anchor` include (but are not limited to):

- `CENTER` - the component will be centered
- `NORTH` - the component will be placed at the top and centered horizontally
- `NORTHEAST` - the component will be placed at the top right corner
- `EAST` - the component will be placed at the left and centered vertically
- `SOUTHEAST` - the component will be placed at the bottom right corner
- `SOUTH` - the component will be placed at the bottom and centered horizontally
- `SOUTHWEST` - the component will be placed at the bottom left corner
- `WEST` - the component will be placed at the left and centered vertically
- `NORTHWEST` - the component will be placed at the top left corner.

Figure 12-37 shows the results of setting `weightx` and `weighty` to 1, `fill` to `NONE` and `anchor` to `SOUTHEAST` in the constraints for the “Tuning” button. Again, you will have to drag the window to a larger size for the effects to be manifested.

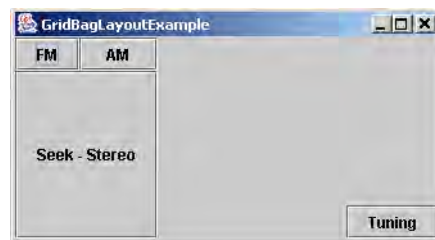


Figure 12-37: GridBagLayoutExample GUI Variation 5

DEFAULT GridBagCONSTRAINTS VALUES

If a `GridBagConstraints` field is not explicitly set, it is given a default value as listed in table 12-8.

Field	Default Value
<code>int gridx</code>	<code>GridBagConstraints.RELATIVE</code>
<code>int gridy</code>	<code>GridBagConstraints.RELATIVE</code>
<code>int gridwidth</code>	1

Table 12-8: GridBagConstraints Fields and Their Default Values

int gridheight	1
double weightx	0.0
double weighty	0.0
int anchor	GridBagConstraints.CENTER
int fill	GridBagConstraints.NONE
Insets insets	new Insets(0, 0, 0, 0)
int ipadx	0
int ipady	0

Table 12-8: GridBagConstraints Fields and Their Default Values

Dialogue with a Skeptic

Skeptic: *[looking over the code in example 12.6 and raising his/her eyebrows]* Why did you use the same GridBagConstraints instance for all the buttons even though each button needs a different set of constraints? I mean, it seems that line 26 which sets the constraints for the “AM” button would also change the constraints object with which the “FM” button was previously associated in line 24. Don’t the “FM” and “AM” buttons both end up with the same constraint values?

Author: No they don’t, but you’ve made a good observation. The only reason that they don’t is that GridBagLayout clones the GridBagConstraints parameter that is passed in with each component. It actually associates this *clone* with the component — not the one you pass in. Without this cloning behavior, *all* the buttons in example 12.6 would have shared the same GridBagConstraints object and the GridBagLayout might have attempted to place each button in exactly the same space as the others.

Skeptic: Relying on GridBagLayout to use clones seems kind of risky to me. Is this cloning behavior documented somewhere?

Author: You’re right. In the absence of some sort of guarantee, relying on a method to clone its parameters is risky. But it is documented. The API for GridBagLayout.addLayoutComponent(Component, Object) states: “Adds the specified component to the layout, using the specified constraints object. Note that constraints are mutable and are, therefore, cloned when cached.”

Skeptic: But, we didn’t call GridBagLayout.addLayoutComponent anywhere in our program!

Author: Yes, that’s true, but we did call Container.add(Component, Object). If you look at the javadocs for any of Container’s add methods, you’ll find this sentence: “This is a convenience method for addImpl(java.awt.Component, java.lang.Object, int).” Now, if you look at the javadocs for addImpl(Component, Object, int), you will see this sentence: “This method also notifies the layout manager to add the component to this container’s layout using the specified constraints object via the addLayoutComponent method.”

Skeptic: Wow, I guess it pays to be adept at using the javadocs!

COMBINING LAYOUT MANAGERS USING JPANELS

Because containers are actually components, they can contain containers as well as components. This simple inheritance structure makes it possible to create arbitrarily complex GUIs by using containers to organize components into groups and subgroups. JPanel is the one Swing component that is specifically intended to be used as a container. It’s useful primarily for organizing a set of components into a single group governed by its own layout manager. Example 12.7 uses a JPanel with GridLayout as the center component in a content pane with BorderLayout. Figure 12-38 shows the results of running example 12.7 and figure 12-39 shows the console output.

12.7 chap12.CombinedLayoutsExample.java

```

1      package chap12;
2      import java.awt.BorderLayout;
3      import java.awt.Container;
4      import java.awt.GridLayout;
5
6      import javax.swing.JButton;
```

```

7      import javax.swing.JFrame;
8      import javax.swing.JPanel;
9
10     import utils.TreePrinterUtils;
11
12     public class CombinedLayoutsExample extends JFrame {
13
14         public CombinedLayoutsExample() {
15             super("CombinedLayoutsExample");
16             setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17             createGUI();
18         }
19
20         public void createGUI() {
21             Container contentPane = getContentPane();
22             contentPane.setLayout(new BorderLayout());
23             contentPane.add(new JButton("WEST"), BorderLayout.WEST);
24             contentPane.add(new JButton("NORTH"), BorderLayout.NORTH);
25             JPanel gridPanel = new JPanel(new GridLayout(4, 3));
26             for (int row = 0; row < 4; ++row) {
27                 for (int col = 0; col < 3; ++col) {
28                     gridPanel.add(new JButton("" + row + " " + col));
29                 }
30             }
31             contentPane.add(gridPanel, BorderLayout.CENTER);
32         }
33
34         public static void main(String[] arg) {
35             JFrame frame = new CombinedLayoutsExample();
36             frame.pack();
37             frame.setVisible(true);
38             TreePrinterUtils.printLayout(frame.getContentPane());
39         }
40     }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap12/CombinedLayoutsExample.java
java -cp classes chap12.CombinedLayoutsExample

```

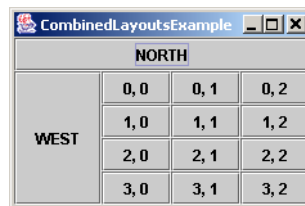


Figure 12-38: CombinedLayoutsExample GUI

Quick Review

Layout managers position components in containers and automatically reposition them as conditions change. All containers can be assigned a layout manager through the method `setLayout(LayoutManager)`. There are two layout manager interfaces defined by the AWT: `LayoutManager` which positions components based on the order in which they are added to the container, and `LayoutManager2` which positions them according to constraints objects. `FlowLayout` and `GridLayout` are examples of the former type. `FlowLayout` arranges components in a container similarly to the way a text processor arranges text in a page. `GridLayout` divides the container into a grid of uniformly sized cells and places components one per cell. `BorderLayout` and `GridBagLayout` are constraints-based layout managers. `BorderLayout` takes up to five components. It allows the component placed in the center to fill all available space as the container resizes, while the other components are given only as much space as is necessary. `GridBagLayout` accepts a `GridBagConstraints` object on a per component basis, encapsulating many options for how each component should be placed in its container. There are several other layout managers including `BoxLayout`, `CardLayout` and `SpringLayout`, and it is even possible to write your own by implementing the `LayoutManager` or `LayoutManager2` interface yourself.

```

JPanel [0, 0, 230, 130] (BorderLayout: 3 children)
+--JButton [0, 26, 68, 104]
+--JButton [0, 0, 230, 26]
+--JPanel [68, 26, 162, 104] (GridLayout: 12 children)
  +--JButton [0, 0, 54, 26]
  +--JButton [54, 0, 54, 26]
  +--JButton [108, 0, 54, 26]
  +--JButton [0, 26, 54, 26]
  +--JButton [54, 26, 54, 26]
  +--JButton [108, 26, 54, 26]
  +--JButton [0, 52, 54, 26]
  +--JButton [54, 52, 54, 26]
  +--JButton [108, 52, 54, 26]
  +--JButton [0, 78, 54, 26]
  +--JButton [54, 78, 54, 26]
  +--JButton [108, 78, 54, 26]

```

Figure 12-39: CombinedLayoutExample Console Output

THE COMPONENTS

So far, we have learned how to create a window, how to place components in a window, how to position components with layout managers, and how to combine layout managers by organizing components into JPanels. Components provide the visual representation of an application's data and they provide the user with the means to graphically communicate complex instructions to an application. The only components we have used so far are JButton, JLabel, JTextArea and JPanel, but the Swing API offers a great wealth of components to handle just about any need a program could have. All Swing components except for the top-level Swing containers extend from `javax.swing.JComponent` which, as you can see from the inheritance tree in figure 12-40, extends from `Container`. This means, strangely enough, that JButton, JTextField, JLabel, etc. are containers just as much as JPanel is, and can theoretically be used as such. *Should we use them as containers?* Probably not, since this would be working against their intended purpose. If we need a container to which to add components, we should stick with JPanel. But the more courageous of you may want to try adding a JTextArea to a JButton just to see what would happen. Go ahead, I'll wait.

```

java.lang.Object
  java.awt.Component
    java.awt.Container
      javax.swing.JComponent

```

Figure 12-40: JComponent Inheritance Hierarchy

BRIEF DESCRIPTIONS

Tables 12-9 through 12-16 include a brief description of all the components that will be used in this chapter's final program. It attempts to organize them into functional categories, but it should be noted that there is much over-

lap of functionality. For example, while some components are listed as containers, all JComponents are actually Containers. As another example, while the JButton and JMenuItem are the only components listed as initiating actions, almost all of the components can and often do initiate actions.

Component	Brief Description
JWindow	A window.
JFrame	A window with a title, border, window controls and an optional menubar.
JDialog	A window that appears when necessary to notify a user of something and/or to receive user input.
JOptionPane	Technically, JOptionPane is not a top-level container. It is used to create modal Dialogs.

Table 12-9: Top-Level Components For Containing Other Components

Component	Brief Description
JPanel	JPanel is a generic lightweight container that groups components together under the control of its layout manager.
JScrollPane	Contains a component that is typically larger than the display area and provides scrollbars for displaying different portions of the component.
JMenuBar	Displays a list of menus horizontally across the top of a JFrame.
JMenu	A special button type that lives in the menu and displays a list of JMenus or JMenuItem in a pop-up list. It is not actually a container but it functions as one.

Table 12-10: Non Top-Level Components For Containing Other Components

Component	Brief Description
JComboBox	A pop-up list that allows a user to choose from a discrete set of options.
JList	Allows a user to select from a number of items.

Table 12-11: Components that Allow the Selection of a Value from a Discrete Set of Values

Component	Brief Description
JSlider	Lets the user select a value by sliding a knob.
JColorChooser	Allows a user to select a color and returns the selected color to the program.

Table 12-12: Components that Allow the Selection of a Value from a Virtual Continuum of Values

Component	Brief Description
JButton	A button.
JMenuItem	A selectable item in a list of items that pops up when a menu is selected.

Table 12-13: Components that Allow the User to Initiate an Action

Component	Brief Description
JToggleButton	A two-state button. It can be either selected or not selected.
JCheckBox	A checkbox. It can be checked (true) or unchecked (false).
JRadioButton	A two-state button. It can be either selected or not selected. Radio buttons are typically arranged in groups where only one radio button can be selected at a time. As such, they allow the user to select an item from a discrete set of options.

Table 12-14: Components that Represent a Boolean Value

Component	Brief Description
JTextField	A single-line area for displaying and editing text.
JTextArea	A multi-line area for displaying and editing text.
JPasswordField	A single-line area for displaying and editing text where the view indicates something was typed, but does not show the original characters.

Table 12-15: Components for Entering Text

Component	Brief Description
JLabel	Displays a small amount of text and/or an image.
JToolTip	When the mouse lingers over a component for a short time, this can appear with a short message typically explaining the use of the component. Can be thought of as a pop-up label.

Table 12-16: View-Only Components

COMPONENT, CONTAINER, AND JCOMPONENT METHODS

Tables 12-17 through 12-28 list and define the most often used methods available to Component, Container and JComponent, filtering away the more esoteric and advanced methods. Event-related methods were also omitted from the table because they are the subject of the next chapter. As a result, only approximately 25% of Component's methods, 33% of Container's methods and 10% of JComponent's methods are included here. The methods have been grouped into the functional categories of Appearance, Size and Location, Visibility, Containment Hierarchy and Other Properties. Browse through the tables now to become familiar with the available methods.

COMPONENT Method Tables

Method Name and Purpose
public Color getBackground() Gets the background color of this component.
public void setBackground(Color) Sets the background color of this component.
public boolean isBackgroundSet() Returns whether the background color has been explicitly set for this Component.
public Color getForeground() Gets the foreground color of this component.
public void setForeground(Color) Sets the foreground color of this component.
public boolean isForegroundSet() Returns whether the foreground color has been explicitly set for this Component.
public boolean isOpaque() Returns true if this component is completely opaque. It returns false by default.
public Cursor getCursor() Gets the cursor set in the component.
public void setCursor(Cursor) Sets the cursor image to the specified cursor.
public boolean isCursorSet() Returns whether the cursor has been explicitly set for this Component.
public Font getFont() Gets the font of this component.
public void setFont(Font) Sets the font of this component.
public boolean isFontSet() Returns whether the font has been explicitly set for this Component.
public void paint(Graphics) Paints this component.
public void paintAll(Graphics) Paints this component and all of its subcomponents.
public void repaint() Repaints this component.
public void repaint(int x, int y, int width, int height) Repaints the specified rectangle of this component.

Table 12-17: Appearance-Related Component Methods

Method Name and Purpose
public Rectangle getBounds() Gets the bounds of this component in the form of a Rectangle object.
public void setBounds(int x, int y, int width, int height) Moves and resizes this component.
public void setBounds(Rectangle) Moves and resizes this component to conform to the specified bounding rectangle.
public Dimension getSize() Returns the size of this component in the form of a Dimension object.
public void setSize(Dimension) Resizes this component so that it has the specified dimension.
public void setSize(int width, int height) Resizes this component so that it has the specified width and height.
public Dimension getMaximumSize() Gets the maximum size of this component.
public Dimension getMinimumSize() Gets the minimum size of this component.
public Dimension getPreferredSize() Gets the preferred size of this component.
public Point getLocation() Gets the location of the top-left corner of this component.
public void setLocation(int x, int y) Moves this component to a new location.
public void setLocation(Point p) Moves this component to a new location.
public Point getLocationOnScreen() Gets the location of the top-left corner of this component (<i>relative to the screen's coordinate system</i>).
public int getWidth() Returns the current width of this component.
public int getHeight() Returns the current height of this component.
public int getX() Returns the current x-coordinate of the component's origin.
public int getY() Returns the current y-coordinate of the component's origin.
public boolean contains(int x, int y) Checks whether this component "contains" the specified point, where x and y are relative to the coordinate system of this component.

Table 12-18: Size- and Location-Related Component Methods

Method Name and Purpose
public boolean isDisplayable() Determines whether this component is displayable.
public boolean isShowing() Determines whether this component is showing on screen.
public boolean isVisible() Determines whether this component should be visible when its parent is visible.
public void setVisible(boolean b) Shows or hides this component depending on the value of parameter b.

Table 12-19: Visibility-Related Component Methods

Method Name and Purpose
public Container getParent() Gets the parent of this component.

Table 12-20: Containment Hierarchy-Related Component Methods

Method Name and Purpose
public String getName() Gets the name of the component.
public void setName(String name) Sets the name of the component to the specified string.
public boolean isEnabled() Determines whether this component is enabled.
public void setEnabled(boolean b) Enables or disables this component, depending on the value of the parameter.
public boolean isLightweight() Determines whether or not this component is lightweight. Lightweight components don't have native toolkit peers.

Table 12-21: Other Property-Related Component Methods

CONTAINER Method Tables

Method Name and Purpose
public Insets getInsets() Determines the insets of this container, which indicate the size of the container's border.
public LayoutManager getLayout() Gets the layout manager for this container.
public void setLayout(LayoutManager mgr) Sets the layout manager for this container.
public void doLayout() Causes this container to layout its components.

Table 12-22: Appearance-Related Container Methods

Method Name and Purpose
public Component add(Component) Adds the specified component to the end of this container.
public Component add(Component, int) Adds the specified component to this container at the given position.
public void add(Component comp, Object constraints) Adds the specified component to the end of this container with the specified constraints.
public void add(Component comp, Object constraints, int index) Adds the specified component to this container with the specified constraints at the specified index.
public Component add(String name, Component comp) Adds the specified component to this container assigning it the specified name.
public int getComponentCount() Gets the number of components in this container.
public Component[] getComponents() Gets all the components in this container.
public Component getComponent(int n) Gets the nth component in this container.
public void remove(Component comp) Removes the specified component from this container.
public void remove(int index) Removes the component at the specified index from this container.
public void removeAll() Removes all the components from this container.
public Component getComponentAt(int x, int y) Locates the component that contains the specified coordinate relative to the container's coordinate system.
public Component getComponentAt(Point p) Gets the component that contains the specified coordinate.

Table 12-23: Containment Hierarchy-Related Container Methods

public Component findComponentAt(int x, int y) Locates the visible child component that contains the specified position.
public Component findComponentAt(Point p) Locates the visible child component that contains the specified point.
public boolean isAncestorOf(Component c) Checks if the specified component is contained in the component hierarchy of this container.

Table 12-23: Containment Hierarchy-Related Container Methods

JComponent Method Tables

Method Name and Purpose
public void setOpaque(boolean isOpaque) If true, the component should paint every pixel within its bounds.
public Border getBorder() Returns the border of this component or null if no border is currently set.
public void setBorder(Border border) Sets the border of this component.

Table 12-24: Appearance-Related JComponent Methods

Method Name and Purpose
public void setMaximumSize(Dimension maximumSize) Sets the maximum size of this component to the specified Dimension.
public boolean isMaximumSizeSet() Returns true if the maximum size has been set to a non-null value; otherwise returns false.
public void setMinimumSize(Dimension minimumSize) Sets the minimum size of this component to the specified Dimension.
public boolean isMinimumSizeSet() Returns true if the minimum size has been set to a non-null value; otherwise returns false.
public void setPreferredSize(Dimension preferredSize) Sets the preferred size of this component.
public boolean isPreferredSizeSet() Returns true if the preferred size has been set to a non-null value; otherwise returns false.

Table 12-25: Size- and Location-Related JComponent Methods

Method Name and Purpose
public Rectangle getVisibleRect() Returns the Component's "visible rectangle" - the intersection of this component's visible rectangle and all of its ancestors' visible rectangles.

Table 12-26: Visibility-Related JComponent Methods

Method Name and Purpose
public Container getTopLevelAncestor() Returns the top-level ancestor of this component (either the containing Window or Applet), or null if this component has not been added to any container.

Table 12-27: Containment Hierarchy-Related JComponent Methods

Other Properties
public String getToolTipText() Returns the tooltip string that has been set with setToolTipText.
public void setToolTipText(String text) Registers the text to display in a tooltip.

Table 12-28: Other Property-Related JComponent Methods

Quick Review

Components provide the visual representation of an application's data and they provide the user with the means to graphically communicate instructions to an application. All Swing components, except for the top-level Swing containers, extend from `javax.swing.JComponent`. Although all Swing components are technically containers, `JPanel` is the Swing container to which we should add components.

The Final GUI

It is beyond the capacity of a single chapter in a book to fully cover the components, and I have no intention of trying. For the most part, the javadocs (*You have been using them, right?*) are sufficient for describing the many properties and methods of the various components. So, this chapter will conclude by combining the learn-by-example approach with your own ability to use the javadocs to deepen your understanding.

Example 12.8 is a larger code example that creates a `JFrame` containing most of the `JComponents` listed in tables 12-9 through 12-16. `JOptionPane` and `JColorChooser` will be incorporated into the program in the next chapter when we begin to customize the application's behavior. Except for lines 67 - 77 which make brief use of the `List`, `Vector` and `Arrays` class, nothing in the application is beyond the scope of this or previous chapters. You may safely ignore these lines as you read through the code or, if you would like, browse ahead to the chapter on Collections. Please read through the code, then compile and run it. Figure 12-41 shows the results of running example 12.8. Click buttons, type text into the fields, resize the window and generally experiment to get an idea of how much behavior the AWT and the Swing components provide by default. Consult tables 12-9 through 12-16 and figure 12-42 to make sure you know which component is which and what it does. Finally, study the output of the call to `TreePrinterUtils.printLayout` (line 100) and figure 12-43 to solidify your understanding of the application's layout.

12.8 chap12.MainFrame.java

```

1      package chap12;
2
3      import java.awt.BorderLayout;
4      import java.awt.Color;
5      import java.awt.GridLayout;
6      import java.util.Arrays;
7      import java.util.List;
8      import java.util.Vector;
9
10     import javax.swing.ButtonGroup;
11     import javax.swing.JButton;
12     import javax.swing.JCheckBox;
13     import javax.swing.JComboBox;
14     import javax.swing.JFrame;
15     import javax.swing.JLabel;
16     import javax.swing.JList;

```



```

17     import javax.swing.JMenu;
18     import javax.swing.JMenuBar;
19     import javax.swing.JMenuItem;
20     import javax.swing.JPanel;
21     import javax.swing.JPasswordField;
22     import javax.swing.JRadioButton;
23     import javax.swing.JScrollPane;
24     import javax.swing.JSlider;
25     import javax.swing.JTextArea;
26     import javax.swing.JTextField;
27     import javax.swing.JToggleButton;
28     import javax.swing.border.EtchedBorder;
29     import javax.swing.border.LineBorder;
30     import javax.swing.border.TitledBorder;
31
32     import utils.TreePrinterUtils;
33
34     public class MainFrame extends JFrame {
35         protected JPanel displayOptionsPanel;
36         protected JButton bgColorButton;
37         protected JButton defaultColorButton;
38
39         protected JToggleButton lockingToggleButton;
40         protected JTextArea textArea;
41         protected JComboBox fontStyleComboBox;
42         protected JSlider fontSizeSlider;
43         protected JLabel sliderLabel;
44         protected JLabel eventLabel;
45
46         protected JList saladList;
47         protected JTextField chosenItemTextField;
48
49         protected JPasswordField secretCodeField;
50
51         protected JMenuItem menuItem1;
52         protected JMenuItem menuItem2;
53         protected JMenuItem menuItem3;
54         protected JMenuItem menuItem4;
55
56         protected JCheckBox vegetablesCheckBox;
57         protected JCheckBox fruitsCheckBox;
58         protected JCheckBox nutsCheckBox;
59         protected JCheckBox cheesesCheckBox;
60
61         protected JRadioButton titleBorderRadioButton;
62         protected JRadioButton lineBorderRadioButton;
63         protected JRadioButton etchedBorderRadioButton;
64         protected JRadioButton bevelBorderRadioButton;
65         protected JRadioButton noBorderRadioButton;
66
67         protected List vegetables =
68             Arrays.asList(
69                 new String[] { "Tomatoes", "Lettuce", "Cucumbers", "Olives" });
70         protected List fruits =
71             Arrays.asList(new String[] { "Apples", "Oranges", "Dates" });
72         protected List nuts =
73             Arrays.asList(new String[] { "Walnuts", "Almonds", "Peanuts" });
74         protected List cheeses =
75             Arrays.asList(
76                 new String[] { "Jack Cheese", "Cheddar Cheese", "Jalapeno Cheese" });
77         protected Vector saladListItems = new Vector(vegetables);
78
79         public MainFrame() {
80             super("Introduction to Swing GUIs");
81             JMenuBar menuBar1 = createMenuBar();
82             setJMenuBar(menuBar1);
83
84             JPanel contentPane = new JPanel();
85             setContentPane(contentPane);
86             contentPane.setLayout(new BorderLayout());
87
88             JPanel centerPanel = createCenterPanel();
89             contentPane.add(centerPanel, BorderLayout.CENTER);
90
91             JPanel northPanel = createNorthPanel();
92             contentPane.add(northPanel, BorderLayout.NORTH);
93
94             JPanel southPanel = createSouthPanel();
95             contentPane.add(southPanel, BorderLayout.SOUTH);
96
97             setSize(600, 600);

```

```

98     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
99     setVisible(true);
100    TreePrinterUtils.printLayout(this);
101    }
102    private JMenuBar createMenuBar() {
103        JMenuBar menuBar = new JMenuBar();
104        JMenu menu1 = new JMenu("Menu 1");
105        menuBar.add(menu1);
106        JMenuItem menuItem1 = new JMenuItem("Menu Item 1");
107        menu1.add(menuItem1);
108        JMenuItem menuItem2 = new JMenuItem("Menu Item 2");
109        menu1.add(menuItem2);
110        JMenu menu2 = new JMenu("Menu 2");
111        menuBar.add(menu2);
112        JMenuItem menuItem3 = new JMenuItem("Menu Item 3");
113        menu2.add(menuItem3);
114        JMenuItem menuItem4 = new JMenuItem("Menu Item 4");
115        menu2.add(menuItem4);
116        return menuBar;
117    }
118    private JPanel createCenterPanel() {
119        JPanel centerPanel = new JPanel(new BorderLayout());
120        JPanel centerNorthPanel = new JPanel(new GridLayout(0, 2));
121        eventLabel =
122            new JLabel("Textarea events will display here.", JLabel.CENTER);
123        eventLabel.setBorder(new LineBorder(Color.red));
124        eventLabel.setOpaque(true);
125        eventLabel.setBackground(Color.yellow);
126        eventLabel.setForeground(Color.red);
127        centerNorthPanel.add(eventLabel, BorderLayout.CENTER);
128        lockingToggleButton = new JToggleButton("Lock Text Area");
129        centerNorthPanel.add(lockingToggleButton, BorderLayout.WEST);
130        centerPanel.add(centerNorthPanel, BorderLayout.NORTH);
131
132        textArea = new JTextArea(30, 60);
133        textArea.setFont(textArea.getFont().deriveFont((float)24));
134        textArea.setLineWrap(true);
135        textArea.setWrapStyleWord(true);
136        centerPanel.add(new JScrollPane(textArea), BorderLayout.CENTER);
137        return centerPanel;
138    }
139    private JPanel createNorthPanel() {
140        JPanel northPanel = new JPanel(new BorderLayout());
141
142        displayOptionsPanel = createDisplayOptionsPanel();
143        northPanel.add(displayOptionsPanel, BorderLayout.WEST);
144
145        JPanel saladOptionsPanel = createSaladOptionsPanel();
146        northPanel.add(saladOptionsPanel, BorderLayout.CENTER);
147
148        return northPanel;
149    }
150    private JPanel createDisplayOptionsPanel() {
151        displayOptionsPanel = new JPanel();
152        displayOptionsPanel.setBorder(new TitledBorder("Panel Options"));
153
154        JPanel colorButtonPanel = new JPanel(new GridLayout(0, 1));
155        bgColorButton = new JButton("Choose Background Color");
156        bgColorButton.setToolTipText("Click to select background color.");
157        colorButtonPanel.add(bgColorButton);
158        defaultColorButton = new JButton("Default Background Color");
159        defaultColorButton.setToolTipText(
160            "Click to restore background color to its default.");
161        colorButtonPanel.add(defaultColorButton);
162        displayOptionsPanel.add(colorButtonPanel);
163
164        JPanel radioPanel = new JPanel(new GridLayout(0, 1));
165        radioPanel.setBorder(new TitledBorder("Borders"));
166
167        noBorderRadioButton = new JRadioButton("No Border");
168        radioPanel.add(noBorderRadioButton);
169        titleBorderRadioButton = new JRadioButton("TitleBorder");
170        titleBorderRadioButton.setSelected(true);
171        radioPanel.add(titleBorderRadioButton);
172        lineBorderRadioButton = new JRadioButton("LineBorder");
173        radioPanel.add(lineBorderRadioButton);
174        etchedBorderRadioButton = new JRadioButton("EtchedBorder");
175        radioPanel.add(etchedBorderRadioButton);
176        bevelBorderRadioButton = new JRadioButton("BevelBorder");
177        radioPanel.add(bevelBorderRadioButton);
178        ButtonGroup buttonGroup = new ButtonGroup();

```

```

179     buttonGroup.add(noBorderRadioButton);
180     buttonGroup.add(titleBorderRadioButton);
181     buttonGroup.add(lineBorderRadioButton);
182     buttonGroup.add(etchedBorderRadioButton);
183     buttonGroup.add(bevelBorderRadioButton);
184     displayOptionsPanel.add(radioPanel);
185
186     return displayOptionsPanel;
187 }
188 private JPanel createSaladOptionsPanel() {
189     JPanel saladOptionsPanel = new JPanel(new BorderLayout());
190     saladOptionsPanel.setBorder(new TitledBorder("Salad Options"));
191
192     JPanel checkBoxPanel = new JPanel(new GridLayout(0, 1));
193     checkBoxPanel.setBorder(new TitledBorder("Ingredients"));
194     vegetablesCheckBox = new JCheckBox("Vegetables");
195     vegetablesCheckBox.setSelected(true);
196     checkBoxPanel.add(vegetablesCheckBox);
197     fruitsCheckBox = new JCheckBox("Fruits");
198     checkBoxPanel.add(fruitsCheckBox);
199     nutsCheckBox = new JCheckBox("Nuts");
200     checkBoxPanel.add(nutsCheckBox);
201     cheesesCheckBox = new JCheckBox("Cheeses");
202     checkBoxPanel.add(cheesesCheckBox);
203     saladOptionsPanel.add(checkBoxPanel, BorderLayout.WEST);
204
205     saladList = new JList(vegetables.toArray());
206     JScrollPane scrollPanel = new JScrollPane(saladList);
207     saladOptionsPanel.add(scrollPanel, BorderLayout.CENTER);
208
209     chosenItemTextField = new JTextField();
210     saladOptionsPanel.add(chosenItemTextField, BorderLayout.SOUTH);
211     return saladOptionsPanel;
212 }
213 private JPanel createSouthPanel() {
214     JPanel southPanel = new JPanel();
215
216     JPanel fontPanel = new JPanel(new BorderLayout());
217     fontPanel.setBorder(new TitledBorder("Font"));
218     fontStyleComboBox =
219         new JComboBox(new String[] { "Plain", "Bold", "Italic", "Bold+Italic" });
220     fontPanel.add(fontStyleComboBox, BorderLayout.WEST);
221
222     fontSizeSlider = new JSlider(1, 200, 24);
223     fontPanel.add(fontSizeSlider, BorderLayout.CENTER);
224     sliderLabel = new JLabel(String.valueOf(fontSizeSlider.getValue()));
225     fontPanel.add(sliderLabel, BorderLayout.EAST);
226     southPanel.add(fontPanel);
227
228     JPanel codePanel = new JPanel();
229     codePanel.setBorder(new EtchedBorder());
230
231     codePanel.add(new JLabel("Enter Secret Code:"));
232     secretCodeField = new JPasswordField(10);
233     codePanel.add(secretCodeField);
234     southPanel.add(codePanel);
235     return southPanel;
236 }
237 public static void main(String[] arg) {
238     MainFrame frame = new MainFrame();
239 }
240 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap12/MainFrame.java
java -cp classes chap12.MainFrame

```

Following is the console output from running the MainFrame application.

```

MainFrame [0, 0, 600, 600] (BorderLayout: 1 child)
+--JRootPane [4, 24, 592, 572] (JRootPane$RootLayout: 2 children)
+--JPanel [0, 0, 592, 572]
+--JLayeredPane [0, 0, 592, 572] (null: 2 children)
+--JMenuBar [0, 0, 592, 23] (DefaultMenuLayout: 2 children)
| +--JMenu [0, 1, 53, 21]
| +--JMenu [53, 1, 53, 21]
+--JPanel [0, 23, 592, 549] (BorderLayout: 3 children)
+--JPanel [0, 193, 592, 295] (BorderLayout: 2 children)

```

```

| +--JPanel [0, 0, 592, 26] (GridLayout: 2 children)
| | +--JLabel [0, 0, 296, 26]
| | +--JToggleButton [296, 0, 296, 26]
| +--JScrollPane [0, 26, 592, 269] (ScrollPaneLayout$UIResource: 3 children)
| | +--JViewport [1, 1, 574, 266] (ViewportLayout: 1 child)
| | | +--JTextArea [0, 0, 574, 960]
| | | +--JScrollPane$ScrollBar [575, 1, 15, 266] (MetalScrollBarUI: 2 children)
| | | | +--MetalScrollBarButton [0, 251, 15, 15]
| | | | +--MetalScrollBarButton [0, 0, 15, 15]
| | | +--JScrollPane$ScrollBar [0, 0, 0, 0] (MetalScrollBarUI: 2 children)
| | | | +--MetalScrollBarButton [0, 0, 0, 0]
| | | | +--MetalScrollBarButton [0, 0, 0, 0]
+--JPanel [0, 0, 592, 193] (BorderLayout: 2 children)
| +--JPanel [0, 0, 320, 193] (FlowLayout: 2 children)
| | +--JPanel [10, 73, 182, 52] (GridLayout: 2 children)
| | | +--JButton [0, 0, 182, 26]
| | | +--JButton [0, 26, 182, 26]
| | +--JPanel [197, 26, 113, 146] (GridLayout: 5 children)
| | | +--JRadioButton [5, 21, 103, 24]
| | | +--JRadioButton [5, 45, 103, 24]
| | | +--JRadioButton [5, 69, 103, 24]
| | | +--JRadioButton [5, 93, 103, 24]
| | | +--JRadioButton [5, 117, 103, 24]
+--JPanel [320, 0, 272, 193] (BorderLayout: 3 children)
| +--JPanel [5, 21, 99, 147] (GridLayout: 4 children)
| | +--JCheckBox [5, 21, 89, 30]
| | +--JCheckBox [5, 51, 89, 30]
| | +--JCheckBox [5, 81, 89, 30]
| | +--JCheckBox [5, 111, 89, 30]
| +--JScrollPane [104, 21, 163, 147] (ScrollPaneLayout$UIResource: 3 children)
| | +--JViewport [1, 1, 160, 144] (ViewportLayout: 1 child)
| | | +--JList [0, 0, 160, 144] (null: 1 child)
| | | | +--CellRendererPane [0, 0, 0, 0] (null: 1 child)
| | | | | +--DefaultListCellRenderer$UIResource [-160, -18, 0, 0]
| | | +--JScrollPane$ScrollBar [0, 0, 0, 0] (MetalScrollBarUI: 2 children)
| | | | +--MetalScrollBarButton [0, 0, 0, 0]
| | | | +--MetalScrollBarButton [0, 0, 0, 0]
| | | +--JScrollPane$ScrollBar [0, 0, 0, 0] (MetalScrollBarUI: 2 children)
| | | | +--MetalScrollBarButton [0, 0, 0, 0]
| | | | +--MetalScrollBarButton [0, 0, 0, 0]
| | +--JTextField [5, 168, 262, 20]
+--JPanel [0, 488, 592, 61] (FlowLayout: 2 children)
| +--JPanel [20, 5, 311, 51] (BorderLayout: 3 children)
| | +--JComboBox [5, 21, 87, 25] (MetalComboBoxUI$MetalComboBoxLayoutManager: 2 children)
| | | +--MetalComboBoxButton [0, 0, 87, 25]
| | | +--CellRendererPane [0, 0, 0, 0] (null: 1 child)
| | | | +--BasicComboBoxRenderer$UIResource [-61, -18, 0, 0]
| | +--JSlider [92, 21, 200, 25]
| | +--JLabel [292, 21, 14, 25]
+--JPanel [336, 13, 235, 34] (FlowLayout: 2 children)
| +--JLabel [7, 9, 106, 16]
| +--JPasswordField [118, 7, 110, 20]

```

Highlights of the Final GUI

Because of the greater complexity of this program's interface, the code for creating the interface is broken into six methods: `createMenuBar()`, `createCenterPanel()`, `createNorthPanel()`, `createSouthPanel()`, `createDisplayOptionsPanel()` and `createSaladOptionsPanel()`.

In lines 121 - 122, the second parameter to the `JLabel` constructor instructs the `JLabel` to center whatever text it has to display. Our previous programs didn't set this parameter on the `JLabels` so they defaulted to `JLabel.LEFT`.

In line 124, we set the `JLabel`'s "opaque" property to true. `JLabels` are transparent by default, which means that they don't paint their background. This allows whatever is behind to show through. To make the label's background yellow, we must set it to yellow, and then call this line to ensure that its background is painted whenever the label is painted.

Line 133 sets the font of the text area to a 24 point version of whatever the text area's default font was.

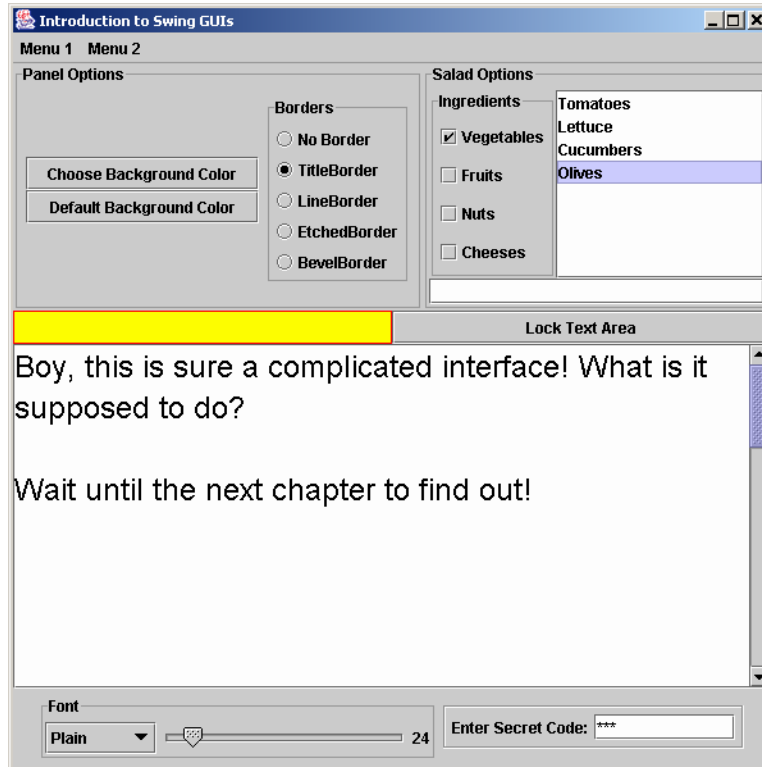


Figure 12-41: MainFrame GUI

Lines 136 and 206 make use of `JScrollPane`s. One `JScrollPane` contains the `textArea` and another contains the list so that when the `textArea` or list contains more text or items than can be displayed, they will become scrollable. `JScrollPane` is a priceless tool, making a difficult task trivial.

In lines 152, 165, 190, 193 and 217, `TitledBorder`s are responsible for the text and gray outline around these panels. Pretty neat, huh?

Lines 178 – 183 utilize a typical pattern for organizing radio buttons. Radio buttons are usually arranged in groups where only one radio button can be selected at a time. Neither a component nor a container, `ButtonGroup` can be thought of as a helper class that gives this typical behavior to a group of radio buttons or any other two-state buttons.

In the `JSlider` constructor (line 222), the first two integer parameters determine the minimum and maximum values the slider can have. The third integer parameter sets the slider's initial value.

In line 224, we initialize the label's text to match the slider's initial value. In the next chapter, sliding the slider will actually update the label.

The integer parameter in the `JPasswordField` constructor (line 232) sets the number of columns of the field. `JPasswordField` inherits the `columns` property from `JTextField`. Surprisingly, the number of columns has no effect on how many characters the field can contain. It affects the preferred size of the field but not in a very accurate manner. Often the number of columns is a moot point because a layout manager will usually determine the size of its components. In this application, however, the `textfield`'s container uses `FlowLayout` which, as you may recall, always sets a component to its preferred size. So, it was important to set the number of columns to a sensible value.

LAYOUT OF THE FINAL GUI

For the first time, we used `TreePrinterUtils.printLayout()` to print the entire window contents rather than just the content pane's contents. In this expanded (*and complete*) textual view of the window's layout, the content pane is listed on the sixth line. It is the second child of a `JLayeredPane`, which is the second child of a `JRootPane`, which is the only child of the `JFrame`. Until now, this chapter intentionally presented a simplified view of the structure of top-level Swing containers. You may never need to know more, but if you would like a complete explanation, please con-

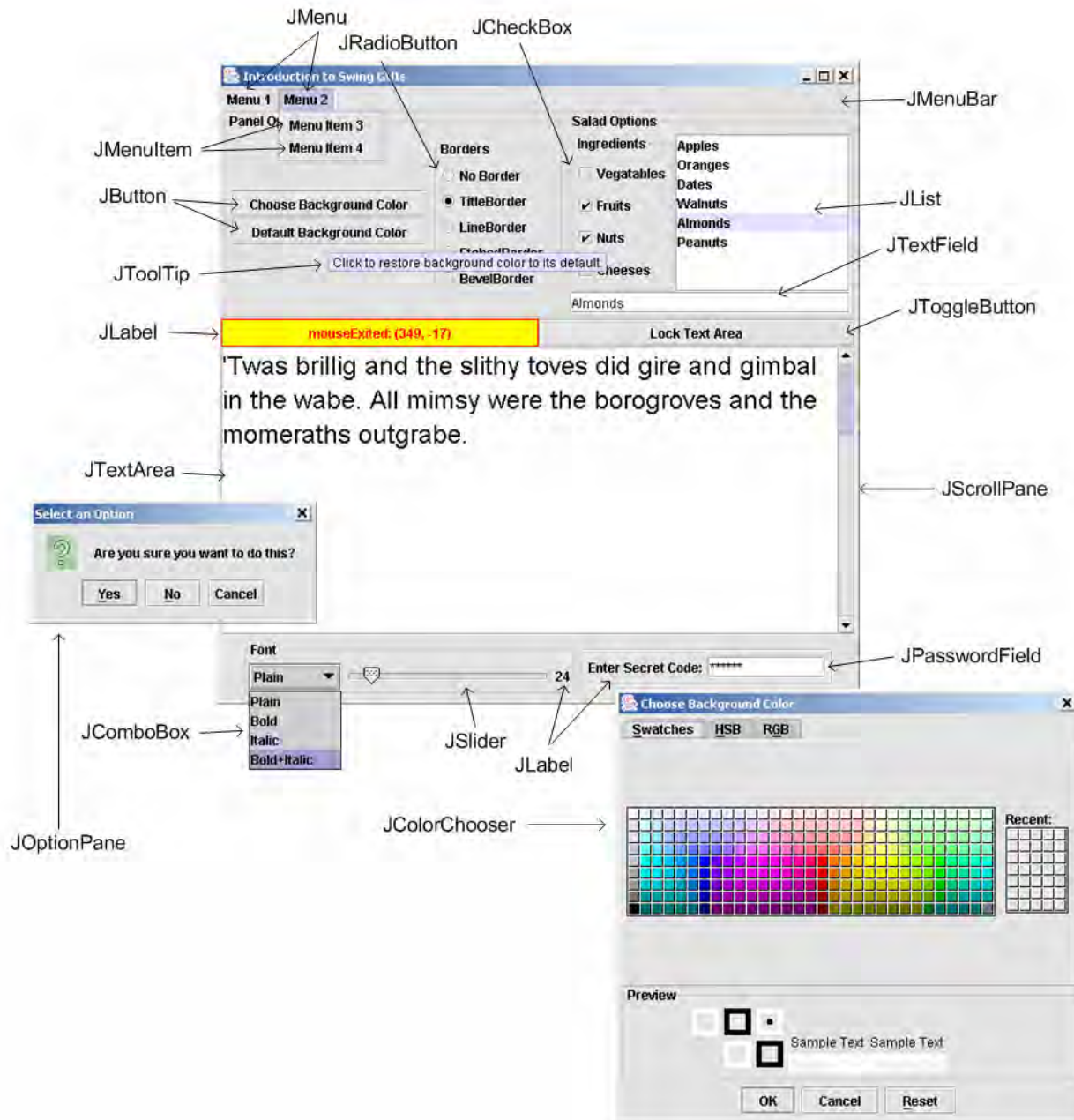


Figure 12-42: Visual Guide to the Components in MainFrame

sult the javadocs for `JRootPane`. Also of interest in the textual view is the structure of the `JMenuBar` which doesn't show the four `JMenuItem`s we added (*Why not?*). Finally take a look at the structure of the `JScrollPane`s and the structure of the `JComboBox`.

The program only uses `BorderLayout`, `FlowLayout` and `GridLayout`, but through the use of multiple nested `JPanel`s, these were enough to produce a complex interface. Components that will be related to each other in the next chapter were generally placed in the same `JPanel`. The basic logic for which layout manager to use in each `JPanel` was:

1. If only one of the components in a group should resize when the container resizes, `BorderLayout` was chosen and that component was placed in the `CENTER` position. The `JPanel` including the `JTextArea` was an obvious candi-

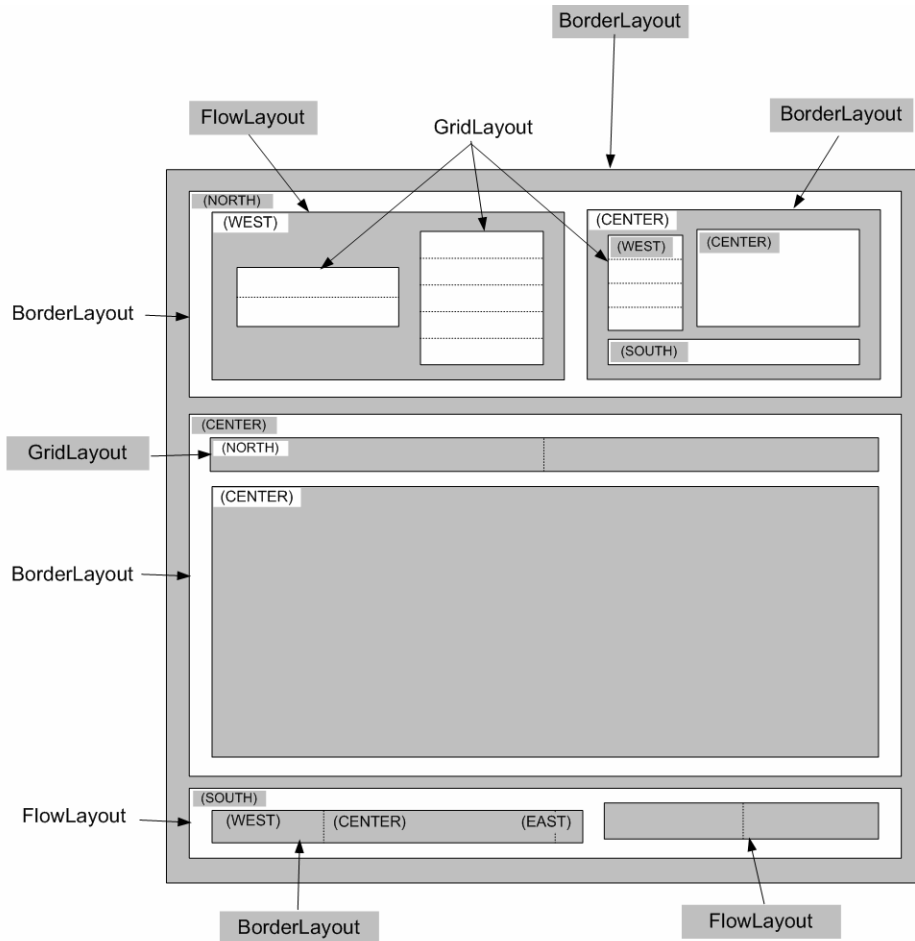


Figure 12-43: MainFrame Layout

date for `BorderLayout`. `BorderLayout` happens to be my favorite layout manager due to its simplicity of use and the way it handles the `CENTER` component. You can see that I used it in several containers.

2. If the components in a group should be sized uniformly then `GridLayout` was chosen. `GridLayout` was an obvious choice for `JRadioButton` and `JCheckBox` groups.

3. If the components were to remain at their preferred sizes and not resize when the container is resized, `FlowLayout` was chosen. `FlowLayout` is not one of my favorites but it had its usefulness in a couple panels, too.

Technically and artistically, there are many ways to design an interface. From the technical perspective, I chose multiple containers and simple layout managers because it seemed like an elegant solution to me. However, the entire GUI could possibly have been constructed from one container and a very complicated `GridBagLayout`. From the artistic perspective, of course, beauty and usability are subjective qualities and I make no claims to the interface's beauty or usability. Its purpose was to put as many different components into one window as seemed reasonable for a chapter on GUIs, and to serve as the basis for the next chapter. The interface meets this objective. If you don't like it artistically, design your own. (I really mean that).

SUMMARY

A *component* is any GUI object that has a visible representation on the screen. A *container* is a component that can “contain” other components. A *top-level* container is a container that exists on the computer screen and can't be contained by another container. Java GUI applications must start with a top-level container otherwise known as a

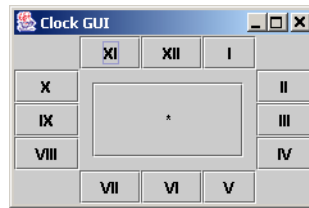
window. There are three types of windows: Window, Frame and Dialog. Their Swing counterparts are JWindow, JFrame and JDialog. All Swing top-level containers have a *content pane* to which all GUI components must be added. JFrame is comprised of a title bar, border, optional menubar and a content pane. Most Swing applications will begin with JFrame since it isn't dependent on the existence of a previously created window.

Layout managers position components in containers and automatically reposition them as conditions change. All containers can be assigned a layout manager through the method `setLayout(LayoutManager)`. There are two layout manager interfaces defined by the AWT: `LayoutManager` which positions components based on the order in which they are added to the container, and `LayoutManager2` which positions them according to *constraints* objects. `FlowLayout` and `GridLayout` are examples of the former type. `FlowLayout` arranges components in a container similarly to the way a text processor arranges text in a page. `GridLayout` divides the container into a grid of uniformly sized cells and places components one per cell. `BorderLayout` and `GridBagLayout` are constraints-based layout managers. `BorderLayout` takes up to five components. It allows the component placed in the center to fill all available space as the container resizes while the other components are given only as much space as is necessary. `GridBagLayout` accepts a `GridBagConstraints` object on a per-component basis, encapsulating many options for how each component should be placed in its container. There are many other layout managers and it is even possible to write your own by implementing the `LayoutManager` or `LayoutManager2` interface.

Components provide the visual representation of an application's data, and they provide the user with the means to graphically communicate complex instructions to an application. All Swing components except for the top-level Swing containers extend from `javax.swing.JComponent`. Although all Swing components are technically containers, `JPanel` is the Swing container to which we should add components.

Skill-Building Exercises

1. **Down and Dirty with GridBagLayout:** Using one `GridBagLayout` and 13 buttons only, create the following GUI.



2. **Modeling the Real World:** Using any combination of `JPanels` and layout managers, pick a household appliance and replicate its interface.
3. **Assimilating Your Knowledge:** Changing nothing in the following code except to implement the `createGUI()` method, create an application that starts up looking like figure 12-44 and when stretched horizontally looks like figure 12-45. Notice that in figure 12-45 only the text fields were stretched.

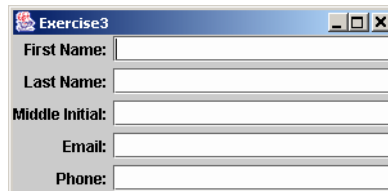


Figure 12-44: Exercise3: Default Size

```

1     package chap12.exercises;
2     import java.awt.*;
3     import javax.swing.*;
4
5     public class Exercise3 extends JFrame{

```

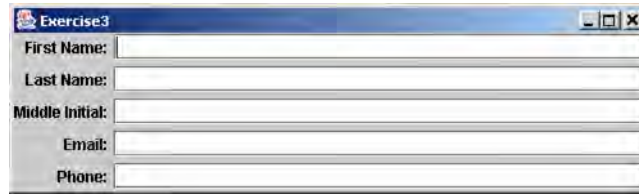



Figure 12-45: Exercise3: Stretched Horizontally

```

6
7     public Exercise3(){
8         super("Exercise3");
9         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10        createGUI();
11    }
12
13    public void createGUI(){
14        //enter code here. Change no other part of this file.
15    }
16
17    public static void main(String[] arg) {
18        JFrame frame = new Exercise3();
19        frame.pack();
20        frame.setVisible(true);
21    }
22 }

```

4. **Exploring a new Layout Manager:** Investigate `BoxLayout` and write a small application that uses it.

SUGGESTED PROJECTS

1. **Borders:** Explore the `javax.swing.border` package and `javax.swing.BorderFactory` class.
2. **Writing Your Own Layout Manager:** Investigate the `LayoutManager` and `LayoutManager2` interfaces. Write an alternative `BorderLayout` class that arranges components according to the diagram in figure 12-46.

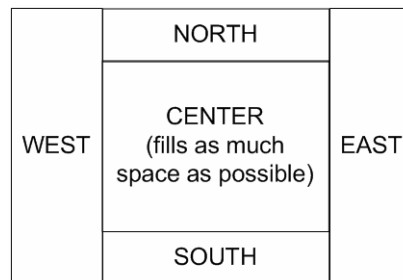


Figure 12-46: Alternate BorderLayout

3. **Specialized Containers:** `JSplitPane` and `JTabbedPane` are specialized containers for holding more than one component. Investigate them and write an application that uses them.

SELF-TEST QUESTIONS

1. Explain the difference between the computer screen's coordinate system and a typical algebraic Cartesian system.

2. What is the difference between `LayoutManager2` and `LayoutManager`?
3. What is the difference between a component and a container?
4. Are Swing components (`JComponents`) containers? Explain.
5. Which `JComponents` allow the user to enter text?
6. Which `JComponents` represent boolean values?
7. Which `JComponents` allow the user to select from a discrete number of options?
8. What are the three Swing top-level containers and what are their differences?
9. Which layout managers automatically resize the components they contain?
10. List and describe the various fields of the `GridBagConstraints` object.
11. Why would one use a layout manager?
12. How is a `JMenu` like a `JComboBox`?
13. Name two layout managers that layout components based on the order in which they were added to a container.
14. Name two constraints-based layout managers.

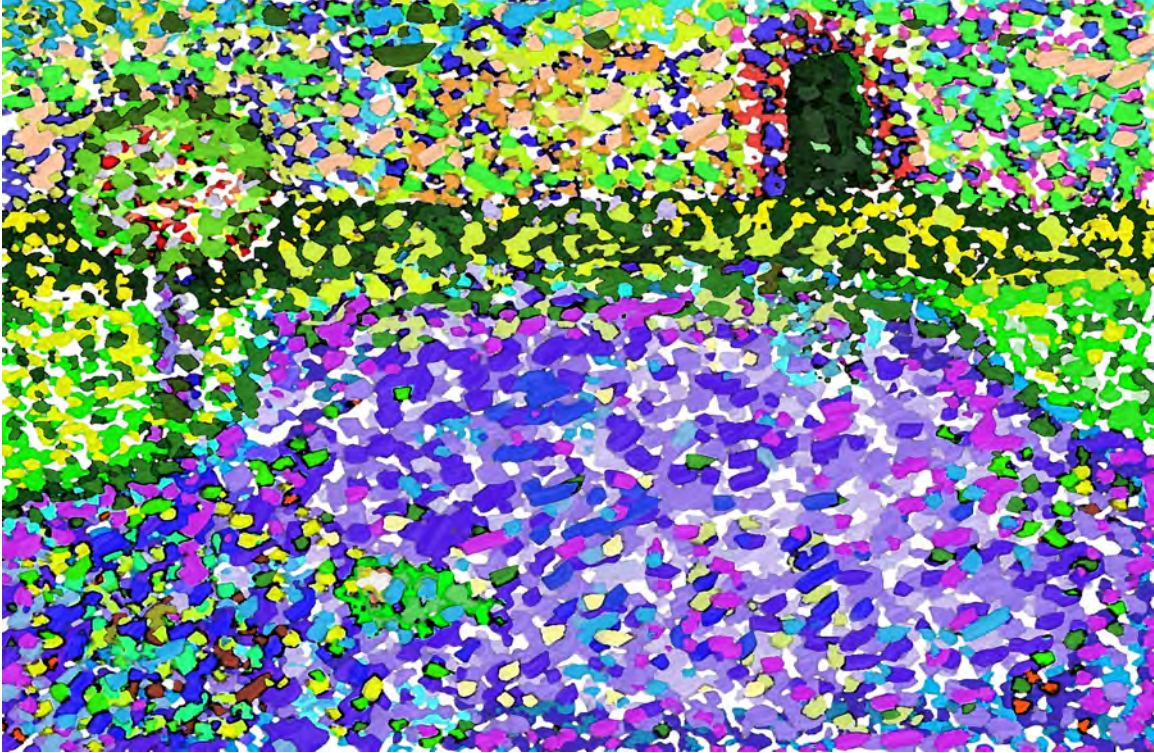
REFERENCES

Java 2 API Specification: [java.sun.com/j2se/1.4.2/docs/api]

The Java Tutorial: [java.sun.com/docs/books/tutorial]

NOTES

CHAPTER 13



GARDEN POND

HANDLING GUI EVENTS

LEARNING OBJECTIVES

- *LIST AND DESCRIBE THE DIFFERENT TYPES OF EVENT LISTENER INTERFACES*
- *LIST AND DESCRIBE THE STEPS REQUIRED TO REGISTER AN EVENT LISTENER WITH A GUI OBJECT*
- *DEMONSTRATE YOUR ABILITY TO HAVE ONE CLASS HANDLE EVENTS GENERATED BY OBJECTS IN ANOTHER CLASS*
- *USE SEVERAL DIFFERENT DESIGN APPROACHES TO CREATE AND REGISTER EVENT LISTENERS*
- *HANDLE EVENTS WITH INNER AND ANONYMOUS CLASSES*
- *CREATE CUSTOM EVENT HANDLERS USING THE ACTIONEVENT, MOUSEEVENT, KEYEVENT, CHANGEVENT, LISTSELECTIONEVENT CLASSES*
- *DESCRIBE THE DIFFERENCES BETWEEN LOW-LEVEL AND SEMANTIC EVENTS*

INTRODUCTION

In the previous chapter we created a rather complex interface (*MainFrame*). In this chapter, we'll discuss how to handle GUI events and we'll customize *MainFrame*'s response to user input. *MainFrame*'s interface is already interactive, thanks to the default behavior of the AWT and Swing components. For instance, clicking on a menu causes menu items to appear; clicking on radio buttons and checkboxes toggles them appropriately; and text areas display typed characters. But if that were all they did, our application would still be virtually useless.

To customize our application's behavior further, we need:

- A way of capturing an event we're interested in (*e.g. typing on the keyboard or clicking with the mouse*) whenever it happens on the component we're interested in
- A way to find out the particulars of the event (*e.g. what key was typed or which button was clicked*) so that we know exactly what happened
- A way of telling the application what piece of code (*that we will write*) to execute and under which conditions to execute it

Fortunately for us, Sun Microsystems has provided an event-handling framework that provides all this. Behind the scenes, when an event occurs, the framework figures out which component the event initially occurred on and sends the event to that component. The component then searches through a list of registered *event listeners* for any it may contain that were listening for that kind of event. It notifies each listener, which is then free to query the event and to respond however it has been coded to respond.

Handling a GUI event using the event-handling framework requires the programmer to accomplish three tasks:

- Choose the event and component of interest
- Implement the appropriate event listener, giving it the desired application behavior
- Register the listener with the component of interest

If the programmer does this, then any occurrence of that event type on that component will cause the execution of whatever code was written into the listener.

EVENT-HANDLING BASICS

THE EVENT

Events can be generated many ways, but are often generated by the user through an input device such as a mouse or keyboard. Events can be divided into two groups: *low-level* and *semantic*. Low-level events represent window-system occurrences or low-level input, such as the pressing of a mouse button or the typing of the letter 'a'. Semantic events include anything else like the selection of a menu or the resizing of a window, and they may or may not be triggered by user input. Generally speaking, semantic events are the ones to pay attention to because they represent more "meaningful" concepts.

All event classes extend from the *EventObject* base class which extends directly from *Object* as shown in figure 13-1.

```
java.lang.Object
  java.util.EventObject
```

Figure 13-1: EventObject Inheritance Hierarchy

This chapter will deal exclusively with events that are triggered by user actions, but *EventObjects* are not limited to representing user actions. They may represent any programmer-defined event. *EventObject*'s one and only constructor requires a non-null object considered to be the *source* of the event. In this chapter, the source object will always be a GUI component such as "the *button* the user clicked" or "the *field* in which the user typed". In general, it

may be absolutely any object so long as that object can be considered the source (*however the programmer chooses to define it*) of the event. Table 13-1 lists and describes the significant `EventObject` methods.

Method Name and Purpose
public Object getSource() Returns the source of the event.

Table 13-1: `EventObject` Methods

Events defined by the Java API all follow the naming convention `<XXX>Event` where `<XXX>` uniquely identifies the type of event. There are very many subclasses of `EventObject`. The ones with which we will become acquainted in this chapter are `ActionEvent`, `MouseEvent`, `KeyEvent`, `ChangeEvent` and `ListSelectionEvent`, but the process for handling events is much the same whatever event type you need to handle.

THE EVENT LISTENER

Event listeners are classes that the programmer writes and registers with a component. Event listeners extend from the `java.util.EventListener` interface, which is a “tagging” interface declaring no methods of its own. For every event type, there is an event listener interface that extends `EventListener` and defines whatever methods are appropriate to that particular event type. By convention, each method of an event listener interface is passed an `EventObject` parameter that encapsulates details specific to the event that caused the method to be called. The programmer must implement these `EventListener` methods to realize whatever application behavior he wants.

In general, event listeners follow the naming convention `<XXX>Listener` where `<XXX>` corresponds to the particular event type listened for. The event listeners with which we will become acquainted in this chapter include `ActionListener`, `MouseListener`, `MouseMotionListener`, `MouseWheelListener`, `KeyListener`, `ChangeListener` and `ListSelectionListener`.

REGISTERING AN EVENT LISTENER WITH A COMPONENT

A component may have any number of registered event listeners, and it provides methods for managing them. Registration methods follow the naming convention `add<XXX>Listener` where `<XXX>Listener` is the type of the `EventListener` parameter. Methods for unregistering specific listener types are named `remove<XXX>Listener`, and methods for retrieving an array of all the listeners of a type are named `get<XXX>Listeners`. `java.awt.Component` also provides a generic method named `getListeners`, to which you pass a listener-class parameter to receive an array of listeners of the specified type. Table 13-2 lists `Component`’s `EventListener` management methods.

Method Name and Purpose
public void add<XXX>Listener(<XXX>Listener instance) Register a listener with a component.
public void remove<XXX>Listener(<XXX>Listener instance) Remove a listener from the component’s registered listeners.
<XXX>Listener[] get<XXX>Listeners() Get all the listeners of a certain type.
EventListener[] getListeners(<XXX>Listener class) Get all the listeners of the specified type.

Table 13-2: `Component` Methods for Managing Event Listeners

Using the API

Often, the best way to see what events a component can respond to is to look through that component’s API for method names beginning with the word “add” and ending with the word “Listener”. Remember to consider methods that the component inherits as well as the ones it defines! This will point you to the event listeners that the component handles. Then, if you look at the API for each different event listener you will see one or more methods that take an `EventObject` parameter. The particular type of this `EventObject` parameter will be a type of event to which the component can respond. As an example, table 13-3 lists the various registration methods offered by `JButton` and their associated event listener types and event types. `JButton` can respond to all of these event types. In some cases there is more than one event listener type for a given event type. We will explore this many-to-one relationship a bit when we handle `MouseEvent`s later.

Registration Method	EventListener Parameter	EventObject Type
<code>addActionListener()</code>	<code>ActionListener</code>	<code>ActionEvent</code>
<code>addChangeListener()</code>	<code>ChangeListener</code>	<code>ChangeEvent</code>
<code>addItemListener()</code>	<code>ItemListener</code>	<code>ItemEvent</code>
<code>addAncestorListener()</code>	<code>AncestorListener</code>	<code>AncestorEvent</code>
<code>addPropertyChangeListener()</code>	<code>PropertyChangeListener</code>	<code>PropertyChangeEvent</code>
<code>addVetoableChangeListener()</code>	<code>VetoableChangeListener</code>	<code>PropertyChangeEvent</code>
<code>addContainerListener()</code>	<code>ContainerListener</code>	<code>ContainerEvent</code>
<code>addComponentListener()</code>	<code>ComponentListener</code>	<code>ComponentEvent</code>
<code>addFocusListener()</code>	<code>FocusListener</code>	<code>FocusEvent</code>
<code>addHierarchyBoundsListener()</code>	<code>HierarchyBoundsListener</code>	<code>HierarchyEvent</code>
<code>addHierarchyListener()</code>	<code>HierarchyListener</code>	<code>HierarchyEvent</code>
<code>addInputMethodListener()</code>	<code>InputMethodListener</code>	<code>InputMethodEvent</code>
<code>addKeyListener()</code>	<code>KeyListener</code>	<code>KeyEvent</code>
<code>addMouseListener()</code>	<code>MouseListener</code>	<code>MouseEvent</code>
<code>addMouseMotionListener()</code>	<code>MouseMotionListener</code>	<code>MouseEvent</code>
<code>addMouseWheelListener()</code>	<code>MouseWheelListener</code>	<code>MouseEvent</code>

Table 13-3: `JButton`’s `EventListener` Registration Methods

Choosing the Right Event

Making an application respond to user input is as simple as linking events to listeners, but choosing the correct event to listen for is not always as obvious as you might wish. Let’s take the prototypical case of an “OK” button in a dialog as shown in figure 13-2. To make the button functional, one might reason (*naively*) that you need to trap a `MouseEvent` with a `MouseListener` because buttons are “clicked” with a mouse. This indeed would allow the button to respond to mouse clicks but the button would only be partially functional. Being an “OK” button in a Dialog, the press of the “Enter” key should also trigger the button. One might reason (*again naively*) that you could also write a `KeyListener` that listens for key events, figures out if the enter-key was pressed and then does the same thing that the `MouseListener` did. That would help, but this approach would be missing the big picture. What if five years from now users of the “Acme Palm-Top Telephonic Device” running the “Acme Java-Enabled Operating System” will only be able to trigger buttons by speaking the button’s name into the mouthpiece of a telephone? (*I made this up but it is*

plausible!). A button written today with MouseListeners and KeyListeners would be broken five years from now on this future system.



Figure 13-2: ACME Product Services Confirmation

So, what does it really mean to make this button fully “functional”? Should we have to worry about keeping up with the myriad platform/operating systems now and in the future in a continual effort to keep it working? The answer is an emphatic NO! Someone, of course, must do that work, but not you, the Java programmer. It is the responsibility and hard work of the developers at Sun Microsystems and elsewhere who port Java to the various platforms and operating systems to make sure that each java port conforms to the target platform’s own rules. In this very typical case, we shouldn’t be interested *how* the button was invoked; we just need to know *that* it was invoked. Therefore, rather than listening for low-level events like mouse-clicks and key-presses which are inherently bound to particular methods of invocation, we should listen for a semantic event that corresponds to “invoking” a button. Is there such an event type? Yes. Thanks to the untiring efforts of the developers at Sun Microsystems, there is a semantic event that encapsulates any input mechanism (*mouse, key, voice, etc*) now and in the future for triggering a button as appropriate for each different platform. Please take the time now to study Figure 13-3 which illustrates the event-handling process as it relates to our “OK” button. As the figure illustrates, the event type that covers all methods of invoking this button would be the ActionEvent. We will discuss ActionEvent soon.

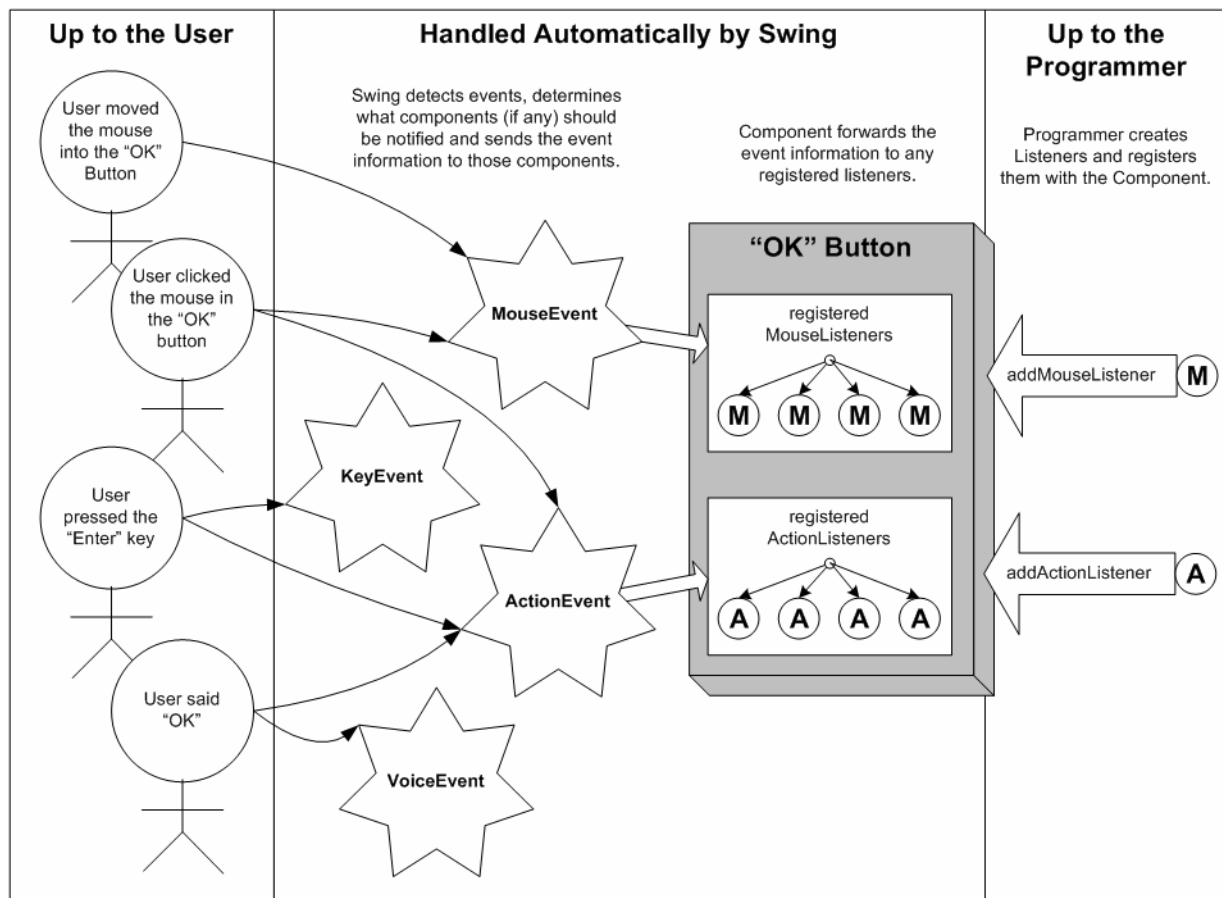


Figure 13-3: Event-Handling Division of Labor

As you can see, while the process for making an application interactive is simple in concept, it pays to know what the API provides when deciding which event type to handle.

Quick Review

Handling GUI events in a Swing application is all about linking events to listeners. Events can be generated many ways but are often generated by the user through an input device such as a mouse or keyboard. The base event class is `EventObject`. Event listeners “listen” or wait for specific events on specific objects and execute whatever code the application developer has written. The base listener class is `EventListener`. To leverage the power of the API and make your code as robust and portable as possible, you should listen for semantic events rather than low-level events whenever possible.

CREATING EVENT LISTENERS

This chapter will take us through the creation of several event listeners as we incrementally add functionality and interactivity to the `MainFrame` application of the previous chapter. Each program in this chapter extends directly or indirectly from `MainFrame`, inheriting functionality from its ancestors as well as providing additional functionality. This use of inheritance is convenient for this chapter’s purposes and for keeping individual source files small, but an equally viable alternative would be to combine all the inherited code into one source file.

```

chap12.MainFrame
  chap13.ListeningMainFrame0
    chap13.ListeningMainFrame1
      chap13.ListeningMainFrame2
        chap13.ListeningMainFrame3
          chap13.ListeningMainFrame4
            chap13.ListeningMainFrame5
              chap13.ListeningMainFrame6
                chap13.ListeningMainFrame7

```

Figure 13-4: Inheritance Hierarchy for the Examples Used in this Chapter

The steps to creating and registering a Listener are always the same and we will follow them with the creation of each class. The first examples will discuss these steps in great detail but as you gain familiarity with the process, we will move quicker and quicker through the steps until, with the creation of the last class, we will breeze quite speedily through the process. Along the way, we will explore various design approaches to implementing an event listener. These design approaches can be applied to any event/listener combination, the choice being determined by factors other than the type of event, listener or component involved.

Steps to Handling an Event:

- Step 1. **Application Behavior:** Determine the desired application behavior from the user’s perspective.
- Step 2. **The Source:** Determine the component that will be the source of the event.
- Step 3. **The Event:** Determine and become familiar with the event type of interest making sure that the source component can respond to it.
- Step 4. **The Listener:** Become familiar with the associated event listener.
- Step 5. **The Design Approach:** Choose a design approach and implement the listener.
- Step 6. **Registration:** Register the listener with the source component.

LISTENINGMAINFRAME0

First we will create the base class for this chapter, `ListeningMainFrame0`, which extends from `MainFrame`. `ListeningMainFrame0` adds no event-handling code. It just customizes the title of the frame and provides an empty

`addListeners()` method which its constructor calls. Each successive subclass will override the `addListeners()` method by calling `super.addListeners()` and adding new event-handling code.

13.1 chap13.ListeningMainFrame0.java

```

1      package chap13;
2
3      public class ListeningMainFrame0 extends chap12.MainFrame {
4
5          public ListeningMainFrame0() {
6              setTitle("Handling GUI Events");
7              addListeners();
8          }
9          protected void addListeners() {}
10         public static void main(String[] arg) {
11             ListeningMainFrame0 frame = new ListeningMainFrame0();
12         }
13     }

```

LISTENINGMAINFRAME1

ListeningMainframe1 will handle the menubar.

STEP 1. APPLICATION BEHAVIOR

When the user selects a menu item from the menu bar, we'll cause a window to appear. It will report which menu item was selected.

STEP 2. THE SOURCE

The source components are the four `JMenuItems`: `menuItem1`, `menuItem2`, `menuItem3` and `menuItem4`.

STEP 3. THE EVENT

Remember that the event must be one to which the component can respond. Table 13-4 lists the `EventListener` registration methods available to `JMenuItem` as well as their associated Event types.

Listener Registration Method	Declaring Class	Corresponding Event Type
<code>addMenuDragMouseListener()</code>	<code>JMenuItem</code>	<code>MenuDragMouseEvent</code>
<code>addMenuKeyListener()</code>	<code>JMenuItem</code>	<code>MenuKeyEvent</code>
<code>addActionListener()</code>	<code>AbstractButton</code>	<code>ActionEvent</code>
<code>addChangeListener()</code>	<code>AbstractButton</code>	<code>ChangeEvent</code>
<code>addItemListener()</code>	<code>AbstractButton</code>	<code>ItemEvent</code>
<code>addAncestorListener()</code>	<code>JComponent</code>	<code>AncestorEvent</code>
<code>addPropertyChangeListener()</code>	<code>JComponent</code>	<code>PropertyChangeEvent</code>
<code>addVetoableChangeListener()</code>	<code>JComponent</code>	<code>PropertyChangeEvent</code>
<code>addContainerListener()</code>	<code>Container</code>	<code>ContainerEvent</code>
<code>addComponentListener()</code>	<code>Component</code>	<code>ComponentEvent</code>
<code>addFocusListener()</code>	<code>Component</code>	<code>FocusEvent</code>
<code>addHierarchyBoundsListener()</code>	<code>Component</code>	<code>HierarchyEvent</code>

Table 13-4: Listeners and Event Types for `JMenuItem`

addHierarchyListener()	Component	HierarchyEvent
addInputMethodListener()	Component	InputMethodEvent
addKeyListener()	Component	KeyEvent
addMouseListener()	Component	MouseEvent
addMouseMotionListener()	Component	MouseEvent
addMouseWheelListener()	Component	MouseEvent

Table 13-4: Listeners and Event Types for JMenuItem

Wow, that’s a lot of event types! Many of them are actually generated and sent to the JMenuItem when a user selects it. So what event type should we listen for when the user selects a menu item? Remember, we shouldn’t be interested in *how* the MenuItem was selected – just *that* it was selected. Referring back to the “Choosing the Right Event” section, we know that we should look for a semantic event. As with the “OK” button in a Dialog, ActionEvent is the event type that will cover all cases when the menu item is selected — whether by mouse press, key press or voice activation.

ACTIONEVENT

Figure 13-5 shows the inheritance hierarchy for the ActionEvent class.

```

java.util.EventObject
    java.awt.AWTEvent
        java.awt.event.ActionEvent
    
```

Figure 13-5: ActionEvent Inheritance Hierarchy

An ActionEvent is a semantic event that indicates that a component-defined action has occurred. Each component that can handle ActionEvents defines for itself what an “action” is. For example, buttons define the click of a mouse to be an action while text fields define the press of the enter-key to be an action. As figure 13-5 shows, ActionEvent extends from AWTEvent, which is the root class for all AWT events. In addition to other methods, AWTEvent defines an “id” property. Event classes that extend AWTEvent define unique ids for each event the EventObject can represent, but it’s usually unnecessary to deal with event ids directly. Table 13-5 lists the one event id defined by ActionEvent.

Event ID and Purpose
public static final int ACTION_PERFORMED = 1001 This indicates that a meaningful action occurred.

Table 13-5: ActionEvent Event IDs

In addition to what it inherits from AWTEvent, the ActionEvent object defines several constants and contains several properties that may be of use in deciding how to handle the event. Some of these are listed in tables 13-6 and 13-7.

Constant Name and Purpose
public static final int SHIFT_MASK An indicator that the shift key was held down during the event.

Table 13-6: ActionEvent Constants

public static final int CTRL_MASK An indicator that the control key was held down during the event.
public static final int META_MASK An indicator that the meta key was held down during the event.
public static final int ALT_MASK An indicator that the alt key was held down during the event.

Table 13-6: ActionEvent Constants

Property Name and Purpose
public String getActionCommand() Returns a string that can be used to identify the particular action to take. The value of this string is often populated automatically. In the case of a button or a menu item it is set to the “text” property of the button or menu item.
public long getWhen() Returns the time the event occurred.
public int getModifiers() Returns the bitwise-or of the modifier constants associated with the modifier keys held down during this action event. These constants are SHIFT_MASK, CTRL_MASK, META_MASK and ALT_MASK. You could use this to see if the control-key or shift-key was pressed with code such as: <pre>int mod = event.getModifiers(); if ((mod & ActionEvent.SHIFT_MASK) != 0) { System.out.println("Shift key was down"); } if ((mod & ActionEvent.CTRL_MASK) != 0) { System.out.println("Control key was down"); }</pre>

Table 13-7: ActionEvent Properties

Components that respond to ActionEvents include AWT’s List, TextField, Button and MenuItem and Swing’s JFileChooser, JComboBox, JTextField, AbstractButton.

STEP 4. THE LISTENER

The associated listener is ActionListener which defines the one method listed in table 13-8. Figure 13-6 shows the inheritance hierarchy for the ActionListener interface.

Method Name and Purpose
public void actionPerformed(ActionEvent) Called whenever an action occurs on the target of the listener.

Table 13-8: ActionListener Methods

```
java.util.EventListener
    java.awt.event.ActionListener
```

Figure 13-6: ActionListener Inheritance Hierarchy

When an action occurs on a component, an `ActionEvent` object is passed as a parameter to the `actionPerformed` method of every `ActionListener` that is registered with the component.

STEP 5: THE DESIGN APPROACH: EXTERNAL CLASS

In example 13.2, we will create an external class called `MyMenuActionListener` that implements `ActionListener`. We will create only one instance of it and add it to all four of our `JMenuItem`s. No matter which `JMenuItem` is selected, the same instance of `MyMenuActionListener` will be notified, so we will have to query the `ActionEvent` for its source to determine which `JMenuItem` was actually selected. Note that in this particular example, the only reason we get the actual `JMenuItem` (line 13) is so that we can invoke `getText()` on it. Invoking the `getActionCommand()` method directly on the `ActionEvent` object would have returned the same value as invoking `getText()` on the `JMenuItem`.

13.2 chap13.MyMenuActionListener.java

```

1      package chap13;
2
3      import java.awt.event.ActionEvent;
4      import java.awt.event.ActionListener;
5
6      import javax.swing.JMenuItem;
7      import javax.swing.JOptionPane;
8
9      public class MyMenuActionListener implements ActionListener {
10
11         public void actionPerformed(ActionEvent e) {
12             Object o = e.getSource();
13             JMenuItem item = (JMenuItem)o;
14             JOptionPane.showMessageDialog(
15                 item,
16                 "You selected \"" + item.getText() + "\".");
17         }
18     }

```

STEP 6: REGISTRATION

In example 13.3, we register the listener with each `JMenuItem` via its `addActionListener()` method.

13.3 chap13.ListeningMainFrame1.java

```

1      package chap13;
2
3      import java.awt.event.ActionListener;
4
5      public class ListeningMainFrame1 extends ListeningMainFrame0 {
6
7         protected void addListeners() {
8             super.addListeners();
9             ActionListener myMenuActionListener = new MyMenuActionListener();
10            menuItem1.addActionListener(myMenuActionListener);
11            menuItem2.addActionListener(myMenuActionListener);
12            menuItem3.addActionListener(myMenuActionListener);
13            menuItem4.addActionListener(myMenuActionListener);
14        }
15        public static void main(String[] arg) {
16            ListeningMainFrame1 frame = new ListeningMainFrame1();
17        }
18    }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap13/ListeningMainFrame1.java
java -cp classes chap13.ListeningMainFrame1

```

LISTENINGMAINFRAME2

With `ListeningMainFrame2`, we will investigate mouse events.

STEP 1: APPLICATION BEHAVIOR

We'll use the event mechanism to capture all mouse activity in the text area and report it in the yellow label that says "Textarea events will display here" (*eventLabel*).

STEP 2: THE SOURCE

The source component is the `JTextArea`, *textArea*.

STEP 3: THE EVENT

The event we need is `MouseEvent` whose inheritance hierarchy is shown in figure 13-7. A `MouseEvent` is a low-level event indicating that a mouse action occurred in a component. All descendants of `java.awt.Component` can respond to `MouseEvents`. This event is used for mouse events (*press, release, click, enter, exit*), mouse motion events (*moves and drags*) and the mouse-wheel event (*the wheel was rotated*). In general, a mouse action is considered to occur in a particular component if and only if the mouse cursor was over a visible portion of the component. Bear in mind that portions of a component may not be visible if other components such as windows or menus are obscuring them.

```
java.awt.event.AWTEvent
  java.awt.event.ComponentEvent
    java.awt.event.InputEvent
      java.awt.event.MouseEvent
```

Figure 13-7: `MouseEvent` Inheritance Hierarchy

`MouseEvent` inherits from `ComponentEvent` the convenience method `getComponent()`, which returns the source of the event as a `Component` (*if it is a Component*) or null otherwise. It inherits the methods `getModifiers()` and `getWhen()` from `InputEvent`, which are functionally equivalent to `ActionEvent`'s methods with the same name. It also inherits some convenience methods from `InputEvent` that are alternatives to using bit-operations on the result of `getModifiers()`. These methods are listed in tables 13-9 through 13-11.

Method Name and Purpose
public Component getComponent() Returns the source of the event if the source is a <code>Component</code> . Otherwise it returns null.

Table 13-9: `ComponentEvent` Methods

Method Name and Purpose
public long getWhen() Functionally equivalent to <code>ActionEvent.getWhen()</code> .
public int getModifiers() Functionally equivalent to <code>ActionEvent.getModifiers()</code> .

Table 13-10: `InputEvent` Methods

Method Name and Purpose
public boolean isAltDown() Returns whether or not the Alt modifier is down on this event.

Table 13-11: `InputEvent` Convenience Methods

public boolean isAltGraphDown() Returns whether or not the Alt-Graph modifier is down on this event.
public boolean isControlDown() Returns whether or not the Control modifier is down on this event.
public boolean isMetaDown() Returns whether or not the Meta modifier is down on this event.
public boolean isShiftDown() Returns whether or not the Shift modifier is down on this event.

Table 13-11: InputEvent Convenience Methods

MouseEvent defines several event ids as listed in table 13-12, each of which identifies a different mouse action.

ID Name and Purpose
public static final int MOUSE_CLICKED = 500 This corresponds to the click of a mouse button. The click of the mouse is a higher level event than the press and release. The user doesn't actually "click" the mouse. The press and release generate a mouse click if the press and release were at the same coordinate (the mouse didn't move in between).
public static final int MOUSE_PRESSED = 501 This corresponds to the press of a mouse button.
public static final int MOUSE_RELEASED = 502 This corresponds to the release of a mouse button. Whether or not it is generated depends on where the mouse was initially pressed. If the mouse was pressed inside the component then a mouse release will be reported even if it was dragged and released outside the component. On the other hand, it will not be reported if the mouse was initially pressed outside the component and then dragged and released inside the component.
public static final int MOUSE_MOVED = 503 This corresponds to the movement of the mouse while no mouse buttons were pressed.
public static final int MOUSE_ENTERED = 504 This indicates that the mouse cursor entered a visible portion of the component.
public static final int MOUSE_EXITED = 505 This indicates that the mouse cursor exited a visible portion of the component.
public static final int MOUSE_DRAGGED = 506 This corresponds to the movement of the mouse while a mouse button was pressed. The event will be generated even when the mouse is outside the component as long as the drag was initiated inside the component.
public static final int MOUSE_WHEEL = 507 This corresponds to the rotation of the mouse wheel.

Table 13-12: MouseEvent Event IDs

MouseEvent defines additional methods that are meaningful specifically to mouse events. These are listed in table 13-13.

Method Name and Purpose
public int getX() Returns the X-coordinate of the event relative to the source component.

Table 13-13: MouseEvent-Specific Methods

public int getY() Returns the Y-coordinate of the event relative to the source component.
public Point getPoint() Returns the X- and Y-coordinates of the event relative to the source component.
public int getClickCount() Returns the number of mouse clicks associated with this event. The number of mouse clicks is associated with mouse presses, releases and clicks. It is zero for other event types.
public int getButton() Returns which, if any, of the mouse buttons has been pressed or released. It will return one of the constant integer values (defined by MouseEvent): BUTTON1, BUTTON2, BUTTON3 or NOBUTTON.

Table 13-13: MouseEvent-Specific Methods

Because the MouseEvent constants BUTTON1, BUTTON2, BUTTON3 are not self explanatory, I prefer to use some SwingUtilities convenience methods for determining which mouse button is pressed. These are listed in table 13-14.

Method Name and Purpose
static boolean isLeftMouseButton(MouseEvent) Returns true if the mouse event specifies the left mouse button.
static boolean isMiddleMouseButton(MouseEvent) Returns true if the mouse event specifies the middle mouse button.
static boolean isRightMouseButton(MouseEvent) Returns true if the mouse event specifies the right mouse button.

Table 13-14: SwingUtilities Helper Methods

STEP 4: THE LISTENER

In order to report all possible mouse event types we will have to implement three listener interfaces. These are MouseListener, MouseMotionListener and MouseWheelListener whose inheritance hierarchy is shown in figure 13-8. Their various method names are obviously derived from the event ids that MouseEvent defines. The event framework automatically calls the correct listener method for the particular event.

```
java.util.EventListener
  java.awt.event.MouseListener
  java.awt.event.MouseMotionListener
  java.awt.event.MouseWheelListener
```

Figure 13-8: MouseListener, MouseMotionListener, and MouseWheelListener Inheritance Hierarchy

MouseListener declares the five methods listed in table 13-15. A MouseEvent is passed as a parameter to each method of a MouseListener that is registered with a component using the addMouseListener() method.

Method	Associated Event ID
public void mousePressed(MouseEvent)	MouseEvent.MOUSE_PRESSED
public void mouseReleased(MouseEvent)	MouseEvent.MOUSE_RELEASED
public void mouseClicked(MouseEvent)	MouseEvent.MOUSE_CLICKED

Table 13-15: MouseListener Methods

<code>public void mouseEntered(MouseEvent)</code>	<code>MouseEvent.MOUSE_ENTERED</code>
<code>public void mouseExited(MouseEvent)</code>	<code>MouseEvent.MOUSE_EXITED</code>

Table 13-15: MouseListener Methods

MouseMotionListener declares the two methods listed in table 13-16. A MouseEvent is passed as a parameter to each method of a MouseMotionListener that is registered with a component using the `addMouseMotionListener()` method.

Method	Associated Event ID
<code>public void mouseMoved(MouseEvent)</code>	<code>MouseEvent.MOUSE_MOVED</code>
<code>public void mouseDragged(MouseEvent)</code>	<code>MouseEvent.MOUSE_DRAGGED</code>

Table 13-16: MouseMotionListener Methods

MouseWheelListener declares the one method listed in table 13-17. A MouseEvent is passed as a parameter to the `mouseWheelMoved()` method of a MouseWheelListener that is registered with a component using the `addMouseWheelListener()` method.

Method	Associated Event ID
<code>public void mouseWheelMoved(MouseEvent)</code>	<code>MouseEvent.MOUSE_WHEEL</code>

Table 13-17: MouseWheelListener Methods

You may be wondering why there isn't just one listener type to handle all mouse events. Well, `MOUSE_MOVED` and `MOUSE_DRAGGED` events were separated into the separate listener type `MouseMotionListener` because tracking the cursor's motion involves significantly more system overhead than tracking other mouse events. And the `MOUSE_WHEEL` event is handled separately by the `MouseWheelListener` because it was first introduced in release 1.4. Just to be complete, there is a fourth Listener named `MouseListener` that extends from both `MouseListener` and `MouseMotionListener`.

STEP 5: THE DESIGN APPROACH: EXTERNAL CLASS WITH FIELDS

In example 13.4, we create the `MyMouseListener` class. In order for it to change the text of `eventLabel`, its methods will need access to `MainFrame`'s `eventLabel`. Since the listener methods are only passed a `MouseEvent` object, we need to provide our listener with an `eventLabel` reference. There are many ways we might do this. In this example, we'll pass the label as a parameter to the `MyMouseListener` constructor so that it can keep a reference to it. If `MainFrame` were to ever remove `eventLabel` or replace it with another component after the construction of this listener, the listener would be stuck with an invalid reference and would no longer function correctly. So, as we develop `MainFrame` we must keep this dependency in mind. We have adopted this simple approach because we do not intend for `MainFrame` to ever change or remove `eventLabel`.

13.4 chap13.MyMouseListener.java

```

1      package chap13;
2
3      import java.awt.event.MouseEvent;
4      import java.awt.event.MouseListener;
5      import java.awt.event.MouseMotionListener;
6      import java.awt.event.MouseWheelEvent;
7      import java.awt.event.MouseWheelListener;
8
9      import javax.swing.JLabel;
10
11     public class MyMouseListener
12     implements MouseListener, MouseMotionListener, MouseWheelListener {
13         JLabel eventLabel;
14

```

```

15     public MyMouseListener(JLabel eventLabel) {
16         this.eventLabel = eventLabel;
17     }
18     public void mouseClicked(MouseEvent e) {
19         displayEvent(e);
20     }
21     public void mousePressed(MouseEvent e) {
22         displayEvent(e);
23     }
24     public void mouseReleased(MouseEvent e) {
25         displayEvent(e);
26     }
27     public void mouseEntered(MouseEvent e) {
28         displayEvent(e);
29     }
30     public void mouseExited(MouseEvent e) {
31         displayEvent(e);
32     }
33     public void mouseDragged(MouseEvent e) {
34         displayEvent(e);
35     }
36     public void mouseMoved(MouseEvent e) {
37         displayEvent(e);
38     }
39     public void mouseWheelMoved(MouseWheelEvent e) {
40         displayEvent(e);
41     }
42     private String getButtonName(MouseEvent e) {
43         int button = e.getButton();
44         if (button == MouseEvent.BUTTON1) {
45             return "Button1";
46         } else if (button == MouseEvent.BUTTON2) {
47             return "Button2";
48         } else if (button == MouseEvent.BUTTON3) {
49             return "Button3";
50         } else {
51             return "Unidentified Button";
52         }
53     }
54     private void displayEvent(MouseEvent e) {
55         String s;
56         int id = e.getID();
57         if (id == MouseEvent.MOUSE_CLICKED) {
58             s = getButtonName(e) + " mouseClicked[" + e.getClickCount() + "];
59         } else if (id == MouseEvent.MOUSE_PRESSED) {
60             s = getButtonName(e) + " mousePressed[" + e.getClickCount() + "];
61         } else if (id == MouseEvent.MOUSE_RELEASED) {
62             s = getButtonName(e) + " mouseReleased[" + e.getClickCount() + "];
63         } else if (id == MouseEvent.MOUSE_ENTERED) {
64             s = "mouseEntered";
65         } else if (id == MouseEvent.MOUSE_EXITED) {
66             s = "mouseExited";
67         } else if (id == MouseEvent.MOUSE_DRAGGED) {
68             s = "mouseDragged";
69         } else if (id == MouseEvent.MOUSE_MOVED) {
70             s = "mouseMoved";
71         } else if (id == MouseEvent.MOUSE_WHEEL) {
72             s = "mouseWheelMoved";
73         } else {
74             s = "Unknown Event";
75         }
76
77         s += " : " + "(" + e.getX() + ", " + e.getY() + ")";
78         System.out.println(s);
79         eventLabel.setText(s);
80     }
81 }

```

The `displayEvent()` method (*lines 54 - 80*) prints event information to the console in addition to displaying it in `eventLabel` because the `mouseClicked` event is generated so quickly after the `mouseReleased` event that the label doesn't have enough time to display the `mouseReleased` event.

STEP 6: REGISTRATION

In `ListeningMainFrame2`, we register the listener with the `JTextArea`. Even though `MyMouseListener` implements all three mouse listener types, a component only reports events to a listener based on which registration meth-

ods are used. So, we must register `myMouseListener` three times. If after running `ListeningMainFrame2` you get tired of all the `mouseMoved` and `mouseDragged` reporting, you can simply comment out line 9 which registers the listener as a `MouseMotionListener`.

13.5 chap13.ListeningMainFrame2

```

1     package chap13;
2
3     public class ListeningMainFrame2 extends ListeningMainFrame1 {
4
5         protected void addListeners() {
6             super.addListeners();
7             MyMouseListener myMouseListener = new MyMouseListener(eventLabel);
8             textArea.addMouseListener(myMouseListener);
9             textArea.addMouseMotionListener(myMouseListener);
10            textArea.addMouseWheelListener(myMouseListener);
11        }
12        public static void main(String[] arg) {
13            ListeningMainFrame2 frame = new ListeningMainFrame2();
14        }
15    }

```

Use the following commands to compile and execute the example. From the directory containing the `src` folder:

```

javac -d classes -sourcepath src src/chap13/ListeningMainFrame2.java
java -cp classes chap13.ListeningMainFrame2

```

LISTENINGMAINFRAME3

With `ListeningMainFrame3`, we will investigate key events. We will also encounter our first application of an inner class.

STEP 1: APPLICATION BEHAVIOR

We'll use the event mechanism to capture keyboard activity in the text area and report it in the same `JLabel` used for reporting mouse events (`eventLabel`).

STEP 2: THE SOURCE

The source component is again the `JTextArea`, `textArea`.

STEP 3: THE EVENT

We are interested in `KeyEvent` whose inheritance hierarchy is shown in figure 13-9. A `KeyEvent` is a low-level event indicating that a key was pressed, released or typed in a component. All descendants of `java.awt.Component` can respond to `KeyEvents`. In order to receive `KeyEvents`, a component must have the keyboard focus. Whether or not a component has keyboard focus is beyond the scope of this chapter to discuss fully. In general, text fields and text areas will have the keyboard focus when they have been clicked with a mouse or when the press of the tab key brings the keyboard focus to them. `KeyEvent` defines the three event ids listed in table 13-18 and the three methods listed in table 13-19.

```

java.awt.AWTEvent
    java.awt.event.ComponentEvent
        java.awt.event.InputEvent
            java.awt.event.KeyEvent

```

Figure 13-9: `KeyEvent` Inheritance Hierarchy

ID Name and Purpose
<p>public static final int KEY_TYPED = 400</p> <p>This KeyEvent is reported to a component when a character is entered. The typing of a character is higher-level than key presses and releases and doesn't generally depend on the platform or keyboard layout. It is often produced by a single key press but it can be produced by a combination of key presses (as in 'shift' + 'a'). Key typed events are only generated for keys that produce Unicode characters. That excludes keys like modifier keys or arrow keys.</p>
<p>public static final int KEY_PRESSED = 401</p> <p>This corresponds to the press of a key. This KeyEvent is generated repeatedly when a key is held down. If the key press signifies a Unicode character such as the letter 'a' for example then a key typed event is also generated. If, on the other hand, the key pressed was the shift-key or some other key that by itself does not signify a Unicode character, then no key typed event is generated.</p>
<p>public static final int KEY_RELEASED = 402</p> <p>This corresponds to the release of a key. Key releases are not usually necessary to generate a key typed event, but there are some cases where the key typed event is not generated until a key is released (e.g., entering ASCII sequences via the Alt-Numpad method in Windows).</p>

Table 13-18: KeyEvent Event IDs

Method Name and Purpose
<p>public char getKeyChar()</p> <p>Returns the character associated with the key in this event. The getKeyChar() method always returns a valid Unicode character or CHAR_UNDEFINED.</p>
<p>public int getKeyCode()</p> <p>Returns the integer keyCode associated with the key in this event. For key pressed and key released events, the getKeyCode method returns the event's keyCode. This value will be one of the many constants with names like VK_XXX which are defined by KeyEvent. Calling this method is the only way to find out about keys that don't generate character input (e.g., action keys, modifier keys, etc.). For key typed events, the getKeyCode() method always returns VK_UNDEFINED.</p>
<p>public int getKeyLocation()</p> <p>Returns the location of the key that originated a key press or key released event. This provides a way to distinguish keys that occur more than once on a keyboard, such as the two shift keys, for example. The possible values are KEY_LOCATION_STANDARD, KEY_LOCATION_LEFT, KEY_LOCATION_RIGHT, KEY_LOCATION_NUMPAD, or KEY_LOCATION_UNKNOWN. This method always returns KEY_LOCATION_UNKNOWN for key-typed events.</p>

Table 13-19: KeyEvent Methods

STEP 4: THE LISTENER

The Listener we need is KeyListener whose inheritance hierarchy is shown in figure 13-10. KeyListener declares the three methods listed in table 13-20 whose names are obviously derived from the event ids that KeyEvent defines. Every time a key event occurs on a component, a KeyEvent object is passed as a parameter to the appropriate method of every KeyListener that is registered with that component.

```
java.util.EventListener
    java.awt.event.KeyListener
```

Figure 13-10: KeyListener Inheritance Hierarchy

Method	Associated Event ID
public void keyTyped(KeyEvent)	KeyEvent.KEY_TYPED
public void keyPressed(KeyEvent)	KeyEvent.KEY_PRESSED
public void keyReleased(KeyEvent)	KeyEvent.KEY_RELEASED

Table 13-20: KeyListener Methods

STEP 5: THE DESIGN APPROACH: INNER CLASS (field-level)

Just as did `MyMouseListener`, `MyKeyListener` will need access to `eventLabel`. Many times, an `EventListener` class needs access to fields or methods of an object (*in our case* `MainFrame`) that would best be left private or protected. If, as is often the case, that listener will only ever be created or referenced by that same object, then it may be preferable to declare the listener as an inner class of the object, thereby giving it automatic access to all the object's fields and simplifying access issues such as pointed out in `MyMouseListener`. So, unlike `MyMouseListener`, `MyKeyListener` will be an inner class of `MainFrame` and will not need any special coding to access `eventLabel`. Nor need it ever worry that `eventLabel` might ever change. It will always have a valid reference to `eventLabel`.

`MyKeyListener` will be an *inner class* of `ListeningMainFrame3`, and `ListeningMainFrame3` will be its *enclosing class*. We will define it at the same level as methods and fields of `ListeningMainFrame3`, effectively making it a property of the `ListeningMainFrame3` object. Consequently, it will have the same accessibility as other class fields and methods of `ListeningFrame3`. By declaring the `MyKeyListener` class to be private, its visibility will be restricted to `ListeningFrame3` only. If it were declared protected, it would be visible only to `ListeningMainFrame3` and its subclasses. If it were declared public or given package protection, then other external classes would be able to reference it.

An inner class is referenced by a non-enclosing class like this: “EnclosingClassName.InnerClassName”. So, if we were to declare `MyKeyListener` as a public inner class (*we won't*), then an external class could contain statements such as

```
ListeningMainFrame3 frame = new ListeningMainFrame3();
ListeningMainFrame3.MyKeyListener keyL = frame.new MyKeyListener();
```

If `ListeningMainFrame3.MyKeyListener` were a static class, then it wouldn't be necessary to have an instance of `ListeningMainFrame3` from which to create the listener instance. Code such as this could be used:

```
ListeningMainFrame3.MyKeyListener keyL =
    new ListeningMainFrame3.MyKeyListener();
```

Within the body of an inner class, the reserved word “this” refers to the inner class instance. An inner class can refer to its enclosing class instance with the notation `<EnclosingClassName>.this`. As an example, `MyKeyListener` can refer to its enclosing `ListeningMainFrame3` class instance as “`ListeningMainFrame3.this`”.

13.6 chap13.ListeningMainFrame3.java

```
1 package chap13;
2
3 import java.awt.event.KeyEvent;
4 import java.awt.event.KeyListener;
5
6 public class ListeningMainFrame3 extends ListeningMainFrame2 {
7
8     protected void addListeners() {
9         super.addListeners();
10        KeyListener myKeyListener = new MyKeyListener();
11        textArea.addKeyListener(myKeyListener);
12    }
13    protected class MyKeyListener implements KeyListener {
14        //implementing KeyListener
```

```

15     public void keyTyped(KeyEvent e) {
16         String display = "keyTyped: " + String.valueOf(e.getKeyChar());
17         System.out.println(display);
18         eventLabel.setText(display);
19     }
20     public void keyPressed(KeyEvent e) {
21         String display = "keyPressed: " + String.valueOf(e.getKeyChar());
22         System.out.println(display);
23         eventLabel.setText(display);
24     }
25     public void keyReleased(KeyEvent e) {
26         String display = "keyReleased: " + String.valueOf(e.getKeyChar());
27         System.out.println(display);
28         eventLabel.setText(display);
29     }
30 }
31 public static void main(String[] arg) {
32     ListeningMainFrame3 frame = new ListeningMainFrame3();
33 }
34 }

```

Event information is printed to the console, in addition to being displayed in eventLabel, because the key released event often follows so quickly after the key typed event that the label doesn't have enough time to display the key typed event.

STEP 6: REGISTRATION

Registering the key listener is handled by line 11 of example13.6.

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap13/ListeningMainFrame3.java
java -cp classes chap13.ListeningMainFrame3

```

Incidentally, although the MyKeyListener class is internal to ListeningMainFrame3, the compiler generates a separate class file for it. Look at the generated class files and you should see a file with a name like ListeningMainFrame3\$MyKeyListener.class corresponding to the inner class! Its name clearly reflects its nested nature.

LISTENINGMAINFRAME4

With ListeningMainFrame4, we will employ another type of inner class.

STEP 1: APPLICATION BEHAVIOR

When the user clicks on the "Choose Background Color" button we will popup a JColorChooser to allow the user to change the background color of one of MainFrame's panels. When the user clicks the "Default Background Color" button, the background will return to its default color.

STEP 2: THE SOURCE

The source components are the two JButtons, bgColorButton and defaultColorButton.

STEP 3: THE EVENT

We could of course choose the MouseEvent, but as you understand by now, it would be better to use the Action-Event.

STEP 4: THE LISTENER

This will be the ActionListener.

STEP 5: THE DESIGN APPROACH: INNER CLASS (LOCAL-VARIABLE-LEVEL)

While we're on the subject of creating inner classes, let's try another way to create an inner class. In the previous approach, we declared `MyKeyListener` at the same level as class fields and methods are declared. We can also declare a class within the body of a method at the local variable scope. A class declared this way is visible only within the body of the method, and neither an external class nor its enclosing class can reference it. The inner class can still refer to its enclosing class in the same manner as do field-level inner classes. We will create an inner class at the local variable level in lines 14 - 32 of example 13.7.

13.7 chap13.ListeningMainFrame4.java

```

1      package chap13;
2
3      import java.awt.Color;
4      import java.awt.event.ActionEvent;
5      import java.awt.event.ActionListener;
6
7      import javax.swing.JColorChooser;
8
9      public class ListeningMainFrame4 extends ListeningMainFrame3 {
10
11         protected void addListeners() {
12             super.addListeners();
13
14             class ColorButtonListener implements ActionListener {
15                 public void actionPerformed(ActionEvent e) {
16                     Object o = e.getSource();
17
18                     if (o == bgColorButton) {
19                         Color color = displayOptionsPanel.getBackground();
20                         color =
21                             JColorChooser.showDialog(
22                                 bgColorButton,
23                                 "Choose Background Color",
24                                 color);
25                         if (color != null) {
26                             displayOptionsPanel.setBackground(color);
27                         }
28                     } else if (o == defaultColorButton) {
29                         displayOptionsPanel.setBackground(null);
30                     }
31                 }
32             };
33
34             ColorButtonListener colorButtonListener = new ColorButtonListener();
35             bgColorButton.addActionListener(colorButtonListener);
36             defaultColorButton.addActionListener(colorButtonListener);
37         }
38         public static void main(String[] arg) {
39             ListeningMainFrame4 frame = new ListeningMainFrame4();
40         }
41     }

```

STEP 6: REGISTRATION

Registration is handled by lines 35 and 36 of example 13.7.

Use the following commands to compile and execute the example: From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap13/ListeningMainFrame4.java
java -cp classes chap13.ListeningMainFrame4

```

The compiler generates a separate class file for any inner class. Look at the generated class files and you should see a file with a name like `ListeningMainFrame4$1$ColorButtonListener.class` corresponding to our inner class.

LISTENINGMAINFRAME5

`ListeningMainFrame5` will add lots of interactivity to our application without creating any additional class at all.

STEP 1: APPLICATION BEHAVIOR

This time we will add the following interactivity to our application.

- The five radio buttons will change the border of one of MainFrame’s JPanels (*displayOption-Panel*).
- The four checkboxes will add or remove items from the JList (*saladList*).
- The combo box will set the font style of the JTextArea (*textArea*).
- The “Lock Text Area” button will toggle the JTextArea’s (*textArea*) editability.
- The “Enter Secret Code” field will compare its text to the JTextField’s (*chosenItemTextField*) text and popup a message.

STEP 2: THE SOURCE

The source components are the five JRadioButtons (*titleBorderRadioButton*, *lineBorderRadioButton*, *etchedBorderRadioButton*, *bevelBorderRadioButton* and *noBorderRadioButton*), the four JCheckBoxes (*vegetablesCheckBox*, *fruitsCheckBox*, *nutsCheckBox* and *cheesesCheckBox*), the JToggleButton (*lockingToggleButton*), the JComboBox (*fontStyleComboBox*) and the JPasswordField (*secretCodeField*).

STEP 3: THE EVENT

We’ll listen for ActionEvents in all cases.

STEP 4: THE LISTENER

We’ll use ActionListener again, of course.

STEP 5: THE DESIGN APPROACH: EXISTING CLASS

Creating an inner class is certainly convenient (*and we will revisit that approach again later with a twist*), but if all we’re doing is implementing an interface, we don’t really need to create a new class at all. We could take any existing class and use it as a listener simply by having it implement whatever EventListener interfaces we need. In this example, we’ll have ListeningMainFrame5 implement ActionListener. Its actionPerformed method will query the ActionEvent object to find out what component generated the event.

13.8 chap13.ListeningMainFrame5.java

```

1      package chap13;
2
3      import java.awt.Color;
4      import java.awt.Font;
5      import java.awt.event.ActionEvent;
6      import java.awt.event.ActionListener;
7      import java.util.List;
8
9      import javax.swing.JCheckBox;
10     import javax.swing.JOptionPane;
11     import javax.swing.border.BevelBorder;
12     import javax.swing.border.EtchedBorder;
13     import javax.swing.border.LineBorder;
14     import javax.swing.border.TitledBorder;
15
16     public class ListeningMainFrame5
17         extends ListeningMainFrame4
18         implements ActionListener {
19
20         protected void addListeners() {
21             super.addListeners();
22
23             titleBorderRadioButton.addActionListener(this);
24             lineBorderRadioButton.addActionListener(this);
25             etchedBorderRadioButton.addActionListener(this);
26             bevelBorderRadioButton.addActionListener(this);
27             noBorderRadioButton.addActionListener(this);
28
29             vegetablesCheckBox.addActionListener(this);
30             fruitsCheckBox.addActionListener(this);

```



```

31     nutsCheckBox.addActionListener(this);
32     cheesesCheckBox.addActionListener(this);
33
34     lockingToggleButton.addActionListener(this);
35
36     fontStyleComboBox.addActionListener(this);
37
38     secretCodeField.addActionListener(this);
39 }
40 public void actionPerformed(ActionEvent e) {
41     Object o = e.getSource();
42
43     //borders
44     if (o == titleBorderRadioButton) {
45         displayOptionsPanel.setBorder(new TitledBorder("Panel Options"));
46         System.out.println(displayOptionsPanel.getInsets());
47     } else if (o == lineBorderRadioButton) {
48         displayOptionsPanel.setBorder(new LineBorder(Color.blue, 10));
49         System.out.println(displayOptionsPanel.getInsets());
50     } else if (o == etchedBorderRadioButton) {
51         displayOptionsPanel.setBorder(new EtchedBorder());
52         System.out.println(displayOptionsPanel.getInsets());
53     } else if (o == bevelBorderRadioButton) {
54         displayOptionsPanel.setBorder(new BevelBorder(BevelBorder.LOWERED));
55         System.out.println(displayOptionsPanel.getInsets());
56     } else if (o == noBorderRadioButton) {
57         displayOptionsPanel.setBorder(null);
58         System.out.println(displayOptionsPanel.getInsets());
59
60         //lock/unlock the textarea
61     } else if (o == lockingToggleButton) {
62         boolean selected = lockingToggleButton.isSelected();
63         textArea.setEditable(!selected);
64         textArea.setOpaque(!selected);
65
66         //update the list
67     } else if (
68         o == vegetablesCheckBox
69         || o == fruitsCheckBox
70         || o == nutsCheckBox
71         || o == cheesesCheckBox) {
72         boolean selected = ((JCheckBox)o).isSelected();
73         List items = null;
74         if (o == vegetablesCheckBox) {
75             items = vegetables;
76         } else if (o == fruitsCheckBox) {
77             items = fruits;
78         } else if (o == nutsCheckBox) {
79             items = nuts;
80         } else if (o == cheesesCheckBox) {
81             items = cheeses;
82         }
83         if (selected) {
84             saladListItems.addAll(items);
85         } else {
86             saladListItems.removeAll(items);
87         }
88         saladList.setListData(saladListItems);
89
90         //change the font style
91     } else if (o == fontStyleComboBox) {
92         int fontStyle = Font.PLAIN;
93         int i = fontStyleComboBox.getSelectedIndex();
94         if (i == 0) {
95             fontStyle = Font.PLAIN;
96         } else if (i == 1) {
97             fontStyle = Font.BOLD;
98         } else if (i == 2) {
99             fontStyle = Font.ITALIC;
100        } else if (i == 3) {
101            fontStyle = Font.BOLD | Font.ITALIC;
102        }
103        textArea.setFont(textArea.getFont().deriveFont(fontStyle));
104
105        //compare secretCodeField to chosenItemTextField
106    } else if (o == secretCodeField) {
107        String message;
108        int messageType;
109        if (secretCodeField
110            .getPassword()
111            .equals(chosenItemTextField.getText())) {

```

```

112         message = "You guessed the secret code!";
113         messageType = JOptionPane.INFORMATION_MESSAGE;
114     } else {
115         message = "That was not the secret code.";
116         messageType = JOptionPane.ERROR_MESSAGE;
117     }
118     JOptionPane.showMessageDialog(
119         secretCodeField,
120         message,
121         "Results",
122         messageType);
123     }
124 }
125 public static void main(String[] arg) {
126     ListeningMainFrame5 frame = new ListeningMainFrame5();
127 }
128 }

```

Using an existing class is perhaps the most efficient way to approach the implementation of an event listener as there is always a bit of overhead with the loading of new class type. However, it can lead to long method definitions, and it tends to clump lots of code together that might be more elegantly expressed separately.

STEP 6: REGISTRATION

Registration is handled by the `addListener` method (*lines 20 - 39*) of example 13.8. It might seem a little strange to pass “this” as the parameter to the `addActionListener()` methods but it’s perfectly legitimate. Once again, the design approach has changed, but we’re still just implementing an `EventListener` interface and registering the implementing object (“this” in this case) with the components.

Use the following commands to compile and execute the example. From the directory containing the `src` folder:

```

javac -d classes -sourcepath src src/chap13/ListeningMainFrame5.java
java -cp classes chap13.ListeningMainFrame5

```

LISTENINGMAINFRAME6

With `ListeningMainFrame6`, we will investigate an event type that does not extend from `AWTEvent`.

STEP 1: APPLICATION BEHAVIOR

The slider in the font panel will set the font size for text in the `JTextArea` (*textArea*).

STEP 2: THE SOURCE

This time the source component is the `JSlider` `fontSizeSlider`.

STEP 3: THE EVENT

`ChangeEvent` is the event of choice. It is definitely a semantic event and is the first event we have dealt with that doesn’t extend from `AWTEvent`. `ChangeEvent` extends directly from `EventObject` (*as shown in figure 13-11*) and defines no methods or fields of its own. It is expected that you would query the object returned by its `getSource()` method or have some other means for discerning what the change is. Components that respond to `ChangeEvents` define for themselves what signifies a “change”. In the case of the `JSlider`, moving the knob generates a `ChangeEvent`. `JComponents` that respond to `ChangeEvents` include `JSlider`, `JProgressBar`, `JSpinner`, `JTabbedPane`, `JViewport` and `AbstractButton`.

```

java.util.EventObject
    javax.swing.event.ChangeEvent

```

Figure 13-11: `ChangeEvent` Inheritance Hierarchy

STEP 4: THE LISTENER

The associated listener is `ChangeListener` whose inheritance hierarchy is shown in figure 13-12. `ChangeListener` defines one method (listed in table 13-21) which is called whenever a change event occurs.

```
java.util.EventListener
    javax.swing.event.ChangeListener
```

Figure 13-12: `ChangeListener` Inheritance Hierarchy

Method Name and Purpose
public void stateChanged(ChangeEvent) Called when the target of the listener has changed its “state”.

Table 13-21: `ChangeListener` Methods**STEP 5: THE DESIGN APPROACH: ANONYMOUS CLASS**

An anonymous class is a class that cannot be referred to – not because of visibility issues but because it is never assigned a name. This is often an attractive approach when it comes to creating listeners. In this example we will implement `ChangeListener` in an anonymous class. Notice how `ListeningMainFrame6` implements the `ChangeListener` interface without ever assigning a class name to it. The variable `myChangeListener` is an instance of a new class whose entire reason for existence is to be assigned to the `myChangeListener` instance variable.

13.9 chap13.ListeningMainFrame6.java

```
1     package chap13;
2
3     import java.awt.Font;
4
5     import javax.swing.JSlider;
6     import javax.swing.event.ChangeEvent;
7     import javax.swing.event.ChangeListener;
8
9     public class ListeningMainFrame6 extends ListeningMainFrame5 {
10
11         protected void addListeners() {
12             super.addListeners();
13             ChangeListener myChangeListener = new ChangeListener() {
14                 public void stateChanged(ChangeEvent e) {
15                     JSlider slider = (JSlider)e.getSource();
16                     int value = slider.getValue();
17                     sliderLabel.setText(String.valueOf(value));
18                     Font font = textArea.getFont();
19                     textArea.setFont(font.deriveFont((float)value));
20                 }
21             };
22
23             fontSizeSlider.addChangeListener(myChangeListener);
24         }
25         public static void main(String[] arg) {
26             ListeningMainFrame6 frame = new ListeningMainFrame6();
27         }
28     }
```

STEP 6: REGISTRATION

Registration is handled by line 23 of example 13.9.

Use the following commands to compile and execute the example. From the directory containing the `src` folder:

```
javac -d classes -sourcepath src src/chap13/ListeningMainFrame6.java
java -cp classes chap13.ListeningMainFrame6
```

Even though our listener class is “anonymous”, the compiler gives it a name when it generates the class file for `ListeningMainFrame6`. If you’re curious, look at the generated class files and you will see a file named something like `ListeningMainFrame6$1.class`.

LISTENINGMAINFRAME7

`ListeningMainFrame7` will introduce another event that doesn’t extend from `AWTEvent`. It will also employ the most syntactically concise design approach.

STEP 1: APPLICATION BEHAVIOR

When the user selects an item in the “Salad Options” list, that item’s name will become the text for the `JTextField` (*chosenItemField*).

STEP 2: THE SOURCE

The source component is the `JList` `saladList`.

STEP 3: THE EVENT

`ListSelectionEvent` is the event of choice. Its inheritance hierarchy is shown in figure 13-13. This semantic event is generated when there has been a change in the current selection of a `JTable` or `JList`. When it is generated, the selection state *may* have changed for one or more rows in the list or table. Similarly to `ChangeEvent`, you have to query the source to obtain more information than the `ListSelectionEvent` instance contains. `ListSelectionEvent` defines the three methods listed in table 13-22.

```
java.util.EventObject
    javax.swing.event.ListSelectionEvent
```

Figure 13-13: `ListSelectionEvent` Inheritance Hierarchy

Method Name and Purpose
public int getFirstIndex() Returns the index of the first row whose selection may have changed.
public int getLastIndex() Returns the index of the last row whose selection may have changed.
public boolean getValuesAdjusting() Returns true if this is one of multiple change events.

Table 13-22: `ListSelectionEvent` Methods

STEP 4: THE LISTENER

The corresponding listener is the `ListSelectionListener` whose inheritance hierarchy is shown in figure 13-14.

```
java.util.EventListener
    javax.swing.event.ListSelectionListener
```

Figure 13-14: `ListSelectionListener` Inheritance Hierarchy

ListSelectionListener defines one method which is called whenever a ListSelectionEvent occurs. This method is listed in table 13-23.

Method Name and Purpose
public void valueChanged(ListSelectionEvent e) Called whenever the value of the selection changes.

Table 13-23: ListSelectionListener Methods

STEP 5: THE DESIGN APPROACH: ANONYMOUS CLASS AND INSTANCE

Finally, in our exploration of design approaches, we will create an anonymous listener class whose instance is also anonymous. We don't need to assign a name to an instance variable if we're only going to refer to it long enough to pass it once as a parameter. So, in this last design approach we create the listener class and its instance both anonymously. This is a wonderfully concise and frequently appropriate design approach.

13.10 chap13.ListeningMainFrame7.java

```

1      package chap13;
2
3      import javax.swing.event.ListSelectionEvent;
4      import javax.swing.event.ListSelectionListener;
5
6      public class ListeningMainFrame7 extends ListeningMainFrame6 {
7
8          protected void addListeners() {
9              super.addListeners();
10             saladList.addListSelectionListener(new ListSelectionListener() {
11                 public void valueChanged(ListSelectionEvent e) {
12                     chosenItemTextField.setText((String) saladList.getSelectedValue());
13                 }
14             });
15         }
16         public static void main(String[] arg) {
17             ListeningMainFrame7 frame = new ListeningMainFrame7();
18         }
19     }

```

STEP 6: REGISTRATION

Registration is handled in place by line 10 of example 13.10. The use of anonymous inner classes can be helpful once you get used to the syntax because the listener instance is defined in the very same place as it is used.

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap13/ListeningMainFrame7.java
java -cp classes chap13.ListeningMainFrame7

```

ADAPTERS

I have chosen to discuss adapters at the end of the chapter, not because they are advanced concepts, but because they are convenience classes doing nothing that couldn't be done by implementing an EventListener interface. Sometimes the situation arises where you need to implement an EventListener interface but are only interested in some of its methods. The Java Language specification of course, does not allow "partial" implementation of interfaces except by abstract classes, so to fulfill the implementation contract, you would have to provide do-nothing implementations for any methods you weren't interested in. An adapter class implements all an interface's methods with do-nothing methods thereby providing you the option of extending it and overriding only the methods in which you are interested. As a concrete example of an adapter, let's look at java.event.MouseAdapter, the source code of which is something like that shown in example 13.11.

13.11 *java.awt.event.MouseAdapter.java*
(extrapolated from Sun's source code)

```

1      package java.awt.event;
2
3      import java.awt.event.MouseEvent;
4      import java.awt.event.MouseListener;
5
6      public abstract class MouseAdapter implements MouseListener {
7          public void mouseClicked(MouseEvent e) {}
8          public void mousePressed(MouseEvent e) {}
9          public void mouseReleased(MouseEvent e) {}
10         public void mouseEntered(MouseEvent e) {}
11         public void mouseExited(MouseEvent e) {}
12     }

```

Now, consider the case where you need to create a `MouseListener`, but are only interested in handling the `MOUSE_CLICKED` event. Below, the `MouseClickedImplementer` class uses the traditional approach of implementing `MouseListener` while the `MouseClickedExtender` class extends from `MouseAdapter`.

13.12 *MouseClickedImplementer.java*

```

1      package chap13;
2
3      import java.awt.event.MouseEvent;
4      import java.awt.event.MouseListener;
5
6      public class MouseClickImplementer implements MouseListener {
7          public void mouseClicked(MouseEvent e) {
8              System.out.println("mouseClicked");
9          }
10         public void mousePressed(MouseEvent e) {}
11         public void mouseReleased(MouseEvent e) {}
12         public void mouseEntered(MouseEvent e) {}
13         public void mouseExited(MouseEvent e) {}
14     }

```

13.13 *MouseClickedExtender.java*

```

1      package chap13;
2
3      import java.awt.event.MouseAdapter;
4      import java.awt.event.MouseEvent;
5
6      public class MouseClickExtender extends MouseAdapter {
7          public void mouseClicked(MouseEvent e) {
8              System.out.println("mouseClicked");
9          }
10     }

```

As you can see, `MouseClickedExtender` is slightly more concise. I find adapters to be useful mostly when I am creating anonymous listener instances and want to keep my code succinct for readability purposes, as in:

```

myComponent.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        System.out.println("mouseClicked");
    }
});

```

SUMMARY

Handling GUI events in a Swing application is all about linking events to listeners. Events can be generated many ways but are often generated by the user through an input device such as a mouse or keyboard. The base event class is `EventObject`. Listeners “listen” for specific events on specific objects and execute whatever code the application developer has written. The base listener class is `EventListener`. There are several design approaches to the linking of events to listeners. These design approaches are:

1. External Class

2. External class with fields
3. Inner class (field level)
4. Inner class (local-variable level)
5. Existing class
6. Anonymous class
7. Anonymous class and instance

In the end, handling GUI events boils down to the same three tasks:

- 1) Choose the event and component of interest;
- 2) Implement the appropriate event listener giving it the desired application behavior;
- 3) Register the listener with the component of interest.

Skill-Building Exercises

1. **Practice with Inner Classes:** Solve the problem of the dangling reference by rewriting `ListeningMainFrame2` using the field-level inner class design approach.
2. **The Existing Class Design:** Rewrite `ListeningMainFrame6` to use itself as the `EventListener` (*existing class design*) instead of using an anonymous class.
3. **Handling the KeyEvent:** Extend `ListeningMainFrame7` to register an additional `KeyListener` with the text area. This `KeyListener` should popup a `JOptionPane` asking if the user needs help when the user “types” Ctrl-H.
4. **Exploring the Complexity of the Event-Handling Framework:** Write a single listener class that implements all the `Listeners` that can be registered with a `JMenuItem`. Implement each method to print the name of the method to the console. Extend `ListeningMainFrame0` and register this listener with `menuItem1` using all the appropriate registration methods. Compile and run the application. Select the menu items and see what is printed to the console. Because this will be a sizable class, I recommend using one of the external class design approaches.
5. **Thinking for Yourself:** Write an application that displays a frame. When the user clicks the close box of the frame, the application should ask the user if he’s sure he wants to quit and offer two options: “Yes” and “No”. Only quit the application if the user responds “yes”. Hint: the source object will be the frame.
6. **Querying the Event:** Write a `MouseListener` that prints a message to the console when the user releases the left mouse button, but only if it has been pressed for at least five seconds. Extend `ListeningMainFrame0` and register this listener with the `JList`.
7. **Venturing Deeper Into the API:** Imagine that the visible area of `ListeningMainFrame0`’s `textArea` is divided into 4 equal quadrants. Keep in mind that `textArea` is contained by a `JScrollPane` and therefore, its visible area is quite different than its bounds. Extend `ListeningMainFrame0` to print a message to the console saying which quadrant was clicked each time the mouse is clicked in the `textArea`.

Suggested Projects

1. **Text Editor:** Write a simple text processor with a text area as its main component. It should have a menu bar and menu items that support cut, copy and paste. Also give it two menu items, “To Upper Case” and “To Lower Case” that take the current text selection and change its case.
2. **Calculator Program Revisited:** Enhance the calculator program, chapter 9, project 4 by creating a GUI for its functionality.

SELF-TEST QUESTIONS

1. What is the base class for all events?
2. What is the base class for all listeners?
3. What are the six steps to handling an event?
4. Explain the difference between a low-level event and a semantic event.
5. Why is it preferable to listen for semantic events?
6. What methods does the `java.util.EventObject` class define?
7. What methods does the `java.util.EventListener` interface declare?
8. What are the benefits and drawbacks to using the “Existing Class” approach to handling an event?
9. What is an inner class? What are the benefits to using an inner class to handle an event?
10. What is an anonymous class? What are the benefits to using an anonymous class to handle an event?
11. What is an adapter? Why might you use an adapter to handle an event?

REFERENCES

Java 2 API Specification: [java.sun.com/j2se/1.4.2/docs/api]

The Java Tutorial: [java.sun.com/docs/books/tutorial]

NOTES

CHAPTER 14



Blurry Subway

AN ADVANCED GUI PROJECT

LEARNING OBJECTIVES

- *UNDERSTAND HOW SWING PAINTS COMPONENTS*
- *UNDERSTAND HOW TO USE THE Graphics class*
- *UNDERSTAND HOW TO LOAD A RESOURCE FROM A PACKAGE-RELATIVE PATH*
- *UNDERSTAND SWING'S SEPARABLE MODEL ARCHITECTURE*
- *UNDERSTAND HOW TO WRITE A CUSTOM RENDERER*
- *CREATE A REUSABLE PLUG-IN RENDERER COMPONENT*
- *UNDERSTAND HOW TO WRITE A CUSTOM EDITOR*
- *DEMONSTRATE YOUR ABILITY TO DEFINE YOUR OWN KIND OF EVENTS AND EVENT LISTENERS*
- *LEARN ONE WAY TO CREATE AN OFFSCREEN IMAGE*
- *PAINT WITH TRANSPARENCY*
- *EXTEND A SWING COMPONENT TO PROVIDE SIGNIFICANT NEW FUNCTIONALITY*
- *GAIN CONFIDENCE MANEUVERING THE WIDE SEAS OF THE SWING API*

INTRODUCTION

This chapter will cover some advanced GUI techniques, but more importantly, it will give you the confidence that you can do some pretty clever programming with the Swing API. The Swing API is powerful and flexible but its complexity can be intimidating. In the hands of an experienced Swing programmer, it can produce marvels, but it takes nothing short of a lot of experience to master. On the other hand, it is possible to write a great variety of useful programs with knowledge only of a small basic subset of its features. What I would like to convey to you in this chapter is the confidence that “if you can imagine it, you can program it with Swing”. To that end, this chapter will guide us through many GUI issues in an attempt to create a GUI interface that transcends the abilities of the standard Swing components.

At this point in your Java studies, I expect that you are able to independently explore issues that a chapter just touches on. For instance, one of the topics this chapter covers is creating offscreen images; but really all it does is show you a couple lines of code that create one type of offscreen image. If you would like to explore the issue deeper (*and it does go deep*) then those two lines will be your stepping stone to further discovery. With that in mind, let us begin our journey.

THE PROJECT

In this chapter, we will develop a small application that is a simple graphic tool a child might use to help reinforce the process of dressing himself. This was indeed the motivation for me when I devised the program. My one child had just turned 5. He knew how to dress himself, but I would often find myself trying to convince him that (*for example*) if he put his shirt on *before* he put on and snapped his pants, it would be easier to tuck it in because his pants wouldn't already be snapped tight. After a while, the boredom of hearing me repeat myself set in and I began to think of a different kind of solution. My mind gravitated to programming and before I knew it, I had an idea for a small Java program that would have a picture of a little boy and a list of all the clothes he could put on. Somehow the program would allow the user to put clothes on the boy. The order in which the clothes were put on would be important, so that it would look funny if, for instance, he tried to put on the socks after he had already put on the shoes. I didn't know exactly how the program's interface would enable the order of clothes to be changed but I knew it was possible. As it turned out, Swing did not offer a component that could do it easily, so I set off on a step by step effort to develop a component that could. I made some mistakes and many discoveries and gradually achieved success. So that this chapter's goal is clearly established, take time now to run the final version of the developed application and play with it.

On the supplemental CD-ROM, navigate to the `Book_Code_Examples/Chapters_12-13-14-16` directory and execute the following command:

```
java -cp classes chap14.gui5.MainFrame
```

Figure 14-1 shows the results of running the application. Select items in the list and click in the checkboxes to cause the boy to wear that article of clothing. Make sure you drag checked list items up and down to see the affect this has on the picture. That's all there is to the program, but it brought a smile to my boys face! This chapter will follow the development of this program from start to finish. In the process we will develop a handy component that I call a `DragList` as well as a reusable customized renderer. But I'm getting ahead of myself. First things first.

THE APPROACH

We will approach our final goal through several steps using different packages to organize the code, reusing as much as we reasonably can. Each step will yield a running program. We will run it and determine what it does that is good, and what we need to add next. Having smaller goals like this helps to build confidence. Just like a climber on a mountain or a tree, it's important to catch our breath every once in a while to appreciate where we are, how far we've come and how far there is to go.

This chapter's program code is organized into six packages named `gui0`, `gui1`, `gui2`, `gui3`, `gui4` and `gui5`, which correspond to six steps in our development process. All the classes for the first step will go into package `gui0`. Then for each additional step, if we can use any class from a previous step as is, we will use it by importing it (*using the import statement*). Any class from a previous step that needs modification will be completely rewritten and placed in



Figure 14-1: This Chapter's Completed Application

the current step's package. Any new class written for the current step will also be put into the current step's package. In the previous chapter, you may remember that `ListeningMainFrame` was repeatedly modified by extension. In this chapter, because class modifications will be more significant, using extension would be awkward at best. So, to sum up, classes that need no modification will be imported from the last package that defined or modified them, while modified and new classes will be part of the current package.

This approach was contrived for the chapter's peculiar purpose of simultaneously minimizing code rewrites and maintaining code from previous steps. If you were developing on your own, you would probably want to keep all your code in one package and modify your classes directly. You might even have a system, as I do, where you archive classes that you are planning to modify just in case the modifications turn sour.

STEP 1 – LAYING THE GROUNDWORK

For now, let's just get a picture of the boy drawn in a window. We'll also create some basic classes that won't need to change as the program develops. These include a class to represent an article of clothing, `Garment`, and a class to handle the drawing of the boy and clothes, `DressingBoard`. We will write the class, `MainFrame`, which will contain the logic for assembling the interface and handling communication between components. `MainFrame` will evolve as the chapter progresses. Lastly, we will take advantage of a utility class for loading resources, `ResourceUtils`.

THE GARMENT CLASS

The `Garment` class exists to encapsulate everything our program needs to know about an individual garment. Since a "garment" for our program's purposes is really just an image drawn on the screen, there are only a few things that our program needs to know about each `Garment`: the image to be drawn, where on the screen to draw it and the name to display in the list. These properties will be *immutable*, meaning they can't change. There is only one more property we would like to know about a `Garment`: is it currently being worn or not? This will determine whether or not to draw it. Due to the nature of our program, this is most definitely a mutable property. You might ask why "worn" should be a property of the `Garment`? Why not a property of the boy for instance? After all, isn't it more natural to ask a boy if he's wearing a shirt than to ask the shirt if it is being worn by a boy? Well, yes it is, but we won't be dealing with a *real* boy or a *real* shirt – only abstractions of them. It will be more convenient to assign the "worn" property to the `Garment` for two reasons: 1) We would not otherwise need a `Boy` object and 2) every `Garment` is either worn or not, so there is a natural one-to-one correspondence. Example 14.1 gives the code for the `Garment` class.

```

1      package chap14.gui0;
2
3      import java.awt.Image;
4
5      public class Garment {
6
7          private Image image;
8          private int x, y;
9          private boolean worn = false;
10         private String name;
11
12         public Garment(Image image, int x, int y, String name) {
13             this.image = image;
14             this.x = x;
15             this.y = y;
16             this.name = name;
17         }
18         public String toString() {
19             return name;
20         }
21         public Image getImage() {
22             return image;
23         }
24         public void setWorn(boolean worn) {
25             this.worn = worn;
26         }
27         public boolean isWorn() {
28             return worn;
29         }
30         public int getX() {
31             return x;
32         }
33         public int getY() {
34             return y;
35         }
36         public String getName() {
37             return name;
38         }
39     }

```

Garment is a simple class that acts as a container for data. It provides getters for all data (*image*, *name*, *worn*, *x*- and *y*-coordinates) and a setter for the mutable data (*worn*). It also overrides the `Object.toString()` method. Later on, you'll see why we needed to override `toString()`.

THE DRESSINGBOARD CLASS

Unlike Garment, DressingBoard is more than just a container of data. It exists to execute the actual drawing of images onto the computer's screen, and contains the logic for how to draw the boy and his clothes in a specified order. Every time the list of worn garments is changed by the user (*through addition, deletion or reordering*), DressingBoard will need to draw the picture again. So, DressingBoard provides a method that accepts an array of Garments and repaints the boy's image accordingly. Our program is not the only entity, however, that will tell the DressingBoard to repaint. The Swing event mechanism, in conjunction with the host operating system, also tells GUI components to repaint themselves. It is for this reason that even though DressingBoard is a simple class to define, we must first delve into the issue of painting in general.

GRAPHICS AND THE AWT/SWING FRAMEWORK

This section just touches on the subject of painting to the extent necessary to understand its use in this chapter's program. It is beyond the current scope to discuss painting in great detail, but much can be learned by experimentation once the basics are understood.

Graphics (The "How" of Drawing)

The Java API offers sophisticated graphics abilities for drawing and painting, encapsulated in the `java.awt.Graphics` class and its descendant, the `java.awt.Graphics2D` class. These two classes provide tools to draw

text, draw and fill shapes and areas of arbitrary complexity, manipulate and transform images, rotate, scale, shear, apply matrix transformations, etc. By the way, the terms “draw” and “paint” are practically synonymous. We will use them interchangeably.

The various methods of the Graphics and Graphics2D classes can be rough-sorted into three categories with the great majority falling into the first two categories:

- Drawing operations that directly change pixel values
- Property-related methods dealing with properties of the Graphics object itself
- Miscellaneous methods

Drawing operation method names tend to begin with words such as “fill” and “draw”. Examples of drawing operations are “drawLine()”, “drawImage()”, “fillPolygon()”, “drawString()” and “fillArc()”. Some have different names such as “clearRect()” and “copyArea()”. In any case, you pass them coordinates, images, polygons, strings, etc. as the names suggest and the methods cause these objects to be drawn or filled or cleared or copied.

Property-related method names tend to begin with “set” and “get” such as “setColor()”, “getColor()”, “setFont()”, “getFont()”, “setPaint()”, “getPaint()”. They set and get various properties of the Graphics object and affect the results of subsequent drawing operations. They do not in themselves do any painting. Some property-setting methods have more active names like “shear()”, “translate()”, “rotate()” and “clip()” but, like their more modestly named cousins, they merely modulate the results of subsequent drawing operations.

Drawing operations and property-related methods are used in conjunction with each other. If, for example, one wanted to draw the string “Aliens” in a Graphics object named g at coordinate 50, 50 with a 48 point, blue sans-serif font, one would call two property-setting methods followed by a single drawing operation:

```
g.setColor(Color.blue);
g.setFont(new Font("SansSerif", Font.PLAIN, 48));
g.drawString("Aliens", 50, 50);
```

The miscellaneous methods include methods that create Graphics objects (*based on the current instance*) and a couple helper methods named hit() and hitClip() that test for shape intersections. We won’t be using any of these miscellaneous methods in this chapter. Figure 14-2 illustrates the function of the Graphics object.

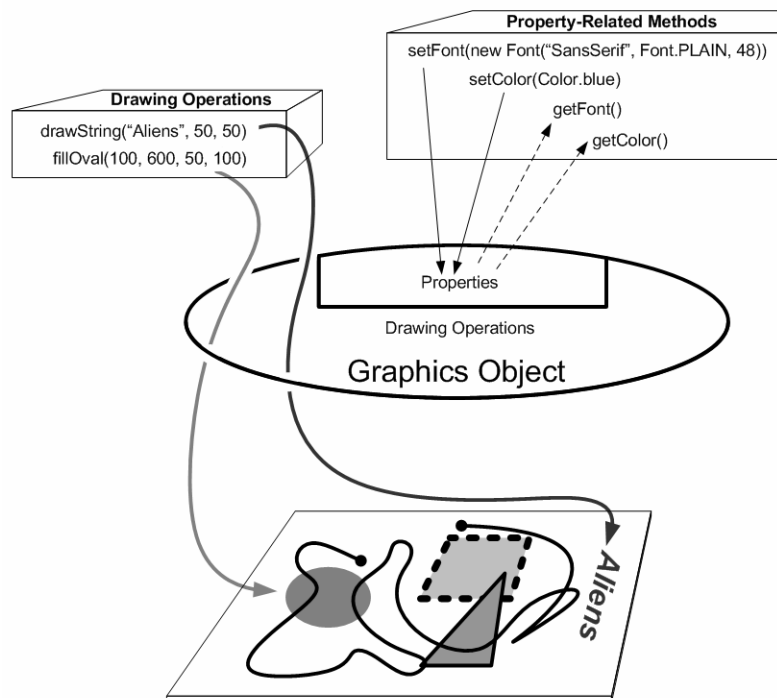


Figure 14-2: Graphics Drawing Operations and Property-Related Methods

The best way to learn how to use the many available Graphics methods is to obtain an instance of a Graphics object and start drawing. But, as you may have noticed if you've peeked at the javadocs, there are no public constructors for Graphics objects. So how does one get started?

THE AWT/SWING FRAMEWORK (THE “WHEN” OF DRAWING)

You will most likely never call a Graphics constructor directly since their constructors are protected. Besides, practically all their methods are abstract. The actual concrete Graphics instance used by the JVM is a platform-specific subclass of Graphics (*or Graphics2D*). In order to obtain a reference to this concrete instance, it is necessary to begin with a reference to either a Component or an Image (*java.awt.image*). You may call the Image.getGraphics() or Component.getGraphics() methods to obtain the Graphics reference that actually draws into them. Components, as you know, are manifested in a visible way on the computer screen. One may draw directly onto the surface of any Component, immediately affecting that component's screen pixels, by obtaining its Graphics and calling drawing operations on it. The Graphics that draws into a Component is known as an *onscreen* Graphics. An Image, on the other hand, is an object residing in memory and, like any other Java object, is not intrinsically viewable. One may draw directly into an Image by getting its Graphics and performing drawing operations on it, but one will only see the results by subsequently copying them into an onscreen Graphics object through one of Graphics' image-copying methods. The Graphics object that draws into an Image is known as an *offscreen* Graphics, and the image into which it draws is referred to as an *offscreen image*.

Before we rush off and start blithely drawing into Components, there is more to say on the subject. Although it is possible at any time to grab a component's Graphics and draw directly in it, you should know that a component's pixels can and will be overwritten by many events not in your control. For instance, when another window comes in front of your lovingly painted component, the pixels that are covered by the window will change to look like the front window and they will need to be restored when the window moves away. The AWT/Swing framework automatically handles the repainting of components “damaged” by system and other events. It can handle cases such as when a window exposes a previously hidden component, as well as other detectable events such as when your application changes the text of a visible JLabel by calling JLabel.setText(). But the framework doesn't automatically restore the results of custom painting, unless you allow it to manage your custom painting code for you. If you do, you'll have to abide by its rules. But don't worry, the framework makes it easy. Just give it the instructions for drawing your component, and it will handle the rest.

WHEN TO PAINT INTO AN AWT COMPONENT

For AWT components, you should override the component's paint() method (*see table 14-1*) and place your custom drawing instructions inside the overridden method body. Component.paint() is called when a component needs to be repainted. You may perform whatever drawing operations you like (*using the provided Graphics*) and relax knowing that whenever your component's pixels get messed up by events beyond your control, your painting instructions will again be executed.

Method Name and Purpose
public void paint(Graphics g) paints the component.

Table 14-1: Component's Painting Method

WHEN TO PAINT INTO A SWING COMPONENT

Swing's JComponents are Components but their paint methods have already been overridden to support the greater power and efficiency of the Swing architecture. For this reason, you are strongly discouraged from overriding the paint() method of a Swing component yourself. When it is time for a Swing component to paint itself, the paint() method for JComponent calls the following three methods in the order listed in table 14-2.

Method Name and Purpose
protected void paintComponent(Graphics g) paints the component
protected void paintBorder(Graphics g) paints the component's border
protected void paintChildren(Graphics g) paints the component's children components

Table 14-2: JComponent's Painting Methods

First, the `paintComponent()` method fills the component with its background color unless the component is transparent (*see* `Component.setOpaque()` and `Component.isOpaque()`). It then performs whatever custom painting is appropriate to the component. Next, the `paintBorder()` method paints whatever border the component may have (*see* `JComponent.setBorder()`). Finally, the `paintChildren()` method calls `Component.paint()` on all the component's children. The method you should override when painting into a `JComponent` is `paintComponent()` which may be considered the Swing counterpart to an AWT component's `paint()` method.

Swing components are *double buffered* by default which means that they have an offscreen `Graphics` as well as an onscreen `Graphics` associated with them. When a Swing component needs to be painted, the AWT/Swing framework performs all the necessary drawing into the offscreen graphics, and then copies the resulting image onto the component's surface. This means that the `Graphics` object provided to the `paintComponent()`, `paintBorder()` and `paintChildren()` methods is not the actual `Graphics` that draws into the component. It is an offscreen `Graphics`, but go ahead and use it as if it were an onscreen `Graphics`. When Swing has executed your code, painted the component's border and finally progressed through the hierarchy of children, it will copy the resulting offscreen image of the component to the screen. Although the use of an offscreen image introduces an additional step into the painting process, there are several benefits to it — including the fact that it is faster.

Calling Repaint

As previously explained, the `paint()` method of a component (*AWT or Swing*) is called when the AWT/Swing framework determines that a component needs to be repainted. It is also called when application code invokes a component's `repaint()` method. (*See tables 14-3 and 14-4*). If at any time during your program's execution you want to repaint a component, presumably because program-specific logic requires the component to display differently, then a call to any one of the repaint methods will cause the Swing painting mechanism to schedule the component to be painted. When it is ready, the framework will call `Component.paint()`. In the case of an AWT component, your code will be executed directly, or in the case of a Swing component, indirectly, when `paint` calls `paintComponent()`. You should never call the `paint()`, `paintComponent()`, `paintBorder()` or `paintChildren()` methods yourself as this could interfere with the framework's automatic painting and cause undesirable results. If the entire component needs repainting, then call the `repaint()` method with no parameters. If, however, only a portion needs repainting, you may be able to make things more efficient by calling one of the `repaint()` methods that take parameters specifying the location and size of a rectangular area to repaint.

Method Name and Purpose
public void repaint() Schedules the component to be repainted.
public void repaint(int x, int y, int width, int height) Schedules the specified rectangle within the component to be repainted.

Table 14-3: Repaint Methods Defined by Component

Method Name and Purpose
public void repaint(Rectangle r) Schedules the specified rectangle within the component to be repainted.

Table 14-4: Repaint Methods Defined by JComponent

Obtaining A Graphics2D

One more painting issue to consider is the use of the Graphics2D object. Graphics2D extends Graphics to provide an additional set of advanced graphics tools such as transparency, anti-aliasing, texture and gradients, as well as more sophisticated control over geometry, coordinate transformations, color management, and text layout. Although the API doesn't explicitly state it, the Graphics object parameter passed to paint(), paintComponent(), paintBorder() and paintChildren() is an instance of Graphics2D. This has been true ever since version 1.2 of the Java platform, so it is safe to cast it to a Graphics2D inside the body of the paint methods as in:

```
protected void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
}
```

THE DRESSINGBOARD CLASS (CONTINUED)

Now that we have discussed painting in the Swing framework, have a look at the code for DressingBoard. (See *example 14.2*.) DressingBoard's constructor loads and stores the image of the boy, calculates once and for all the image's width and height and initializes its private array of Garments to a zero-length array. We give our DressingBoard the method, setOrder(Garment[] garments) which updates its private array of Garments and triggers a call to paint() by calling repaint(). And of course, there is the all-important paintComponent() method which is overridden to paint the background white, do some arithmetic to find out where center is, and then draw the image of the boy followed by each garment in the Garment array (*in order*).

14.2 chap14.gui0.DressingBoard.java

```
1 package chap14.gui0;
2
3 import java.awt.Color;
4 import java.awt.Graphics;
5 import java.awt.Image;
6
7 import javax.swing.BorderFactory;
8 import javax.swing.JComponent;
9 import javax.swing.border.BevelBorder;
10
11 import utils.ResourceUtils;
12
13 public class DressingBoard extends JComponent {
14     private Image boyImage;
15     private Garment[] order;
16     private int boyWidth;
17     private int boyHeight;
18
19     public DressingBoard() {
20         setOpaque(true);
21         setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
22
23         //image of completely clothed boy courtesy of www.hasslefreeclipart.com
24         boyImage = ResourceUtils.loadImage("chap14/images/boy.gif", this);
25
26         boyWidth = boyImage.getWidth(this);
27         boyHeight = boyImage.getHeight(this);
28         setOrder(new Garment[0]);
29     }
30     public void paintComponent(Graphics g) {
31         super.paintComponent(g);
32         int width = getWidth();
33         int height = getHeight();
34         g.setColor(Color.WHITE);
35         g.fillRect(0, 0, width, height);
```

```

36     int offsetX = (width - boyWidth) / 2;
37     int offsetY = (height - boyHeight) / 2;
38     g.drawImage(boyImage, offsetX, offsetY, this);
39     for (int i = 0; i < order.length; ++i) {
40         Garment garment = order[i];
41         g.drawImage(
42             garment.getImage(),
43             garment.getX() + offsetX,
44             garment.getY() + offsetY,
45             this);
46     }
47 }
48 public void setOrder(Garment[] order) {
49     this.order = order;
50     repaint();
51 }
52 }

```

LOADING AN IMAGE OR RESOURCE

Thanks to the introduction of the `javax.imageio` package in Java 1.4, the loading of an image from a URL has become a simple and straightforward process. It wasn't always this easy, as you may see if you look at the methods `loadImage_pre_imageio` and `loadImage_pre_swing` in lines 34 - 48 of example 14.3. On the other hand, constructing the correct URL can still be tricky and this is the reason for the `ResourceUtils.relativePathToUrl(String, Object)` method (*lines 15 - 19*).

14.3 *utils.ResourceUtils.java*

```

1     package utils;
2
3     import java.awt.Component;
4     import java.awt.Image;
5     import java.awt.MediaTracker;
6     import java.awt.Toolkit;
7     import java.awt.image.BufferedImage;
8     import java.io.IOException;
9     import java.net.URL;
10
11    import javax.imageio.ImageIO;
12    import javax.swing.ImageIcon;
13
14    public class ResourceUtils {
15        public static URL relativePathToUrl(String path, Object obj) {
16            Class clazz = obj instanceof Class ? (Class)obj : obj.getClass();
17            URL url = clazz.getClassLoader().getResource(path);
18            return url;
19        }
20        public static BufferedImage loadImage(String path, Object obj) {
21            return loadImage(relativePathToUrl(path, obj));
22        }
23        public static BufferedImage loadImage(URL url) {
24            try {
25                return ImageIO.read(url);
26            } catch (IOException ignore) {}
27
28            return null;
29        }
30
31        /*
32         * For historical interest
33         */
34        public static Image loadImage_pre_imageio(URL url) {
35            return new ImageIcon(url).getImage();
36        }
37        public static Image loadImage_pre_swing(URL url) {
38            Component comp = new Component() {};
39            MediaTracker tracker = new MediaTracker(comp);
40            Image image = Toolkit.getDefaultToolkit().createImage(url);
41            tracker.addImage(image, 0);
42            try {
43                tracker.waitForID(0);
44                return image;
45            } catch (InterruptedException ignore) {}
46
47            return null;
48        }
49    }

```

Ordinarily when it's time to distribute your application, it is wise to package it into a single jar file that includes classes as well as other application resources such as images. Unfortunately, the URL for a resource is significantly different when it's in a jar file than when it's sitting in your computer's file system. If your program code accesses a resource via a hard-coded URL, the program code will have a dependency on the distribution format, but if you specify resource paths that are relative to your application's package structure, you can bundle all your application resources into a single jar file without having to change any code due to URL changes. The `ResourceUtils.loadImage(String, Object)` method allows you to specify class-relative paths from which it dynamically constructs the correct URL. In order for it to do this, it requires an object parameter. If the object is of type `Class` then it must be an application-defined class. If it is not of type `Class`, then it must be an instance of an application-defined class. Usually it's most convenient to pass "this" as the `Object` parameter.

As a concrete example of usage, I keep all my generated class files and resources for this book in `C:/JavaForArtists/classes`. The images for this chapter are located in `C:/JavaForArtists/classes/chap14/images`. When I run `chap14.gui5.MainFrame` using the following command...

```
java -cp C:/JavaForArtists/classes chap14.gui5.MainFrame
```

...the URL for the boy image is...

```
file:/C:/JavaForArtists/classes/chap14/images/boy.gif
```

But if I jar the contents of `C:/JavaForArtists/classes/chap14` and `C:/JavaForArtists/classes/utls` into a file called `Dressing.jar` located in `C:/JavaForArtists` using the following command...

```
jar -cmf C:/JavaForArtists/Dressing.jar -C C:/JavaForArtists/classes utls
-C C:/JavaForArtists/classes chap14
```

...and then run the program using...

```
java -cp C:/JavaForArtists/Dressing.jar chap14.gui5.MainFrame
```

...the URL for the boy image is...

```
jar:file:/C:/JavaForArtists/Dressing.jar!/chap14/images/boy.gif
```

This is quite a bit different from the previous URL. Fortunately, by using the `loadImage(String path, Object obj)` method, all my code needs to specify for the location of the boy image is...

```
chap14/images/boy.gif
```

...as in line 24 of `DressingBoard.java`. The correct URL will be dynamically constructed.

THE MAINFRAME CLASS

`MainFrame` is the entry point for our application. It constructs the main window. As this program evolves, `MainFrame` will also evolve. Each package will define its own `MainFrame` class.

14.4 chap14.gui0.MainFrame.java

```
1     package chap14.gui0;
2
3     import java.awt.BorderLayout;
4     import java.awt.Container;
5
6     import javax.swing.JFrame;
7
8     public class MainFrame extends JFrame {
9
10        private DressingBoard dressingBoard;
```

```

11
12     public MainFrame() {
13         setTitle(getClass().getName());
14         setSize(600, 400);
15         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16
17         Container contentPane = getContentPane();
18         contentPane.setLayout(new BorderLayout());
19         dressingBoard = new DressingBoard();
20         contentPane.add("Center", dressingBoard);
21     }
22     public static void main(String[] arg) {
23         new MainFrame().setVisible(true);
24     }
25 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap14/gui0/MainFrame.java
java -cp classes chap14.gui0.MainFrame

```

Run the program to see the image of a boy. Yikes! We need to put some clothes on him!

Quick Review

We created the Garment and DressingBoard classes which will not need further modification. We wrote MainFrame which we expect to evolve. We looked at various methods of loading an image and a convenience method for dynamically constructing a resource's URL. We learned about the Graphics and Graphics2D classes. We learned how the AWT/Swing framework handles the painting of components and how to override the paint() method of an AWT component or the paintComponent() method of a Swing component to do custom painting.

STEP 2 – SWING'S SEPARABLE MODEL ARCHITECTURE

In this next version of MainFrame, we will add a shuffled list of Garments to a JList at the left side of the window. JList is an example of what is known as Swing's *separable model architecture* which is loosely based on MVC (*model-view-controller*) architecture. We will gain a deeper understanding of JList and how it uses its data model, ListModel. Also, just to make the program a bit more fun at this stage (*and the image more discrete*), we'll have ListSelectionEvents trigger the DressingBoard to draw selected Garments on the boy.

Using A ListModel

One of the differences between AWT's and Swing's approaches to GUI architecture is that Swing employs a separable model architecture while the AWT does not. The difference between Swing's JList and AWT's List is a prime example of this.

If you look at the API for java.awt.List, you will see many methods for managing the data it contains. It provides methods for adding items, removing items and replacing items, for instance. Unfortunately, it requires that the items be Strings. As a result of the tight coupling between List and its data, if you use a List in your program, you are committed to representing your data as a list of strings and to giving the List sole ownership over it.

Let's see what implications the use of a List would have on this chapter's program. Our application's data is a list of Garments. In order to use a List, however, the application would have to maintain a list of strings for the List's benefit as well as the list of Garments. This means that at any time the list of Garments changed, the application would have to make sure that List's list of Strings was changed accordingly, or application data and view would be out of sync. This could be a maintenance headache.

JList, on the other hand, provides no methods for managing data. It maintains view functionality but delegates data management to a separate Object, a ListModel. Hence the term *separable model*. ListModel is an interface that declares a bare minimum of methods to manage a read-only list of data and a set of ListDataListeners. Unlike List, it requires only that the data be comprised of Objects, thus freeing the application to maintain its data however it wishes. Any object implementing the ListModel interface can be shared with any number of JLists through JList's

`getModel()` and `setModel()` methods. Furthermore, since `JList` automatically registers a `ListDataListener` with its `ListModel`, it is notified whenever an invocation of the `ListModel`’s methods changes the data. Table 14-5 lists the methods defined by the `ListModel` interface. Table 14-6 lists the methods available to `JList` for manipulating its `ListModel`.

Let’s see what implications the use of a `JList` would have on this chapter’s program. Our application could maintain a list of `Garments` wrapped in a `ListModel` interface and share it with as many `JLists` as it wanted. This means that with no further effort, all `JLists` would be guaranteed to be in sync with the underlying data. This is definitely a better scenario.

Method Name and Purpose
public int getSize() Returns the number of elements in the list model.
public Object getElementAt(int index) Returns the element at the specified index in the list model.
public void addListDataListener(ListDataListener l) Called automatically when this list model becomes the component’s data model.
public void removeListDataListener(ListDataListener l) Called automatically when this list model is no longer the component’s data model.

Table 14-5: ListModel Methods

Method Name and Purpose
public ListModel getModel() Gets the current <code>ListModel</code> .
public void setModel(ListModel model) Installs the specified <code>ListModel</code> .
public void setListData(Object[] listData) Installs a new <code>ListModel</code> containing the specified data.
public void setListData(Vector listData) Installs a new <code>ListModel</code> containing the specified data.

Table 14-6: JList’s ListModel Methods

THE MAINFRAME CLASS

In this step (*gui1 package*) `MainFrame` creates, initializes and manipulates the `JList`’s `ListModel`. In the `initList` method (*lines 44 - 79*), it creates a `Vector` of `Garments`, shuffles it and passes it to the `JList`, which will create a `ListModel` for itself containing the `Garments` in the shuffled order. This book has not yet formally covered the `Vector` class or the `Collections` utility class used here but if *lines 70 - 77* are not self-explanatory, a quick look into the javadocs for those classes should suffice to explain them. For more information on `Collections` in general please refer to Chapter 17 — `Collections`.

`MainFrame` also implements `ListSelectionListener` and registers itself with the `JList` so that any change in the `JList`’s selected items will trigger a call to `MainFrame`’s `valueChanged()` method. The `valueChanged()` method translates the `JList`’s selection state into the `ListModel`’s data by setting all `Garments` in the `ListModel` to be worn if and only if they are selected in the `JList`. The next logical step is factored into a new method, `redrawBoy()`, which is not dependent on the selection state of the `JList` so that, if we ever need to redraw the boy for other reasons, we will have a convenient method that doesn’t require a `ListSelectionEvent` parameter. `RedrawBoy()` obtains all the garments from the `ListModel` and creates an array of worn garments which it passes to `DressingBoard.setOrder()`. The call in line 90 to `order.toArray()` takes advantage of a feature of `Collections` which will be discussed later. Simply pass in an array of

the appropriate type to the `toArray()` method. It will return an array of that same type containing every element in the collection. Example 14.5 gives the code for the modified `MainFrame` class.

14.5 chap14.gui1.MainFrame.java

```

1      package chap14.gui1;
2
3      import java.awt.BorderLayout;
4      import java.awt.Container;
5      import java.awt.Image;
6      import java.awt.Point;
7      import java.util.Collections;
8      import java.util.Vector;
9
10     import javax.swing.BorderFactory;
11     import javax.swing.JFrame;
12     import javax.swing.JList;
13     import javax.swing.ListModel;
14     import javax.swing.border.BevelBorder;
15     import javax.swing.event.ListSelectionEvent;
16     import javax.swing.event.ListSelectionListener;
17
18     import utils.ResourceUtils;
19     import chap14.gui0.DressingBoard;
20     import chap14.gui0.Garment;
21
22     public class MainFrame extends JFrame implements ListSelectionListener {
23
24         private DressingBoard dressingBoard;
25         private JList garmentList;
26
27         public MainFrame() {
28             setTitle(getClass().getName());
29             setSize(600, 400);
30             setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31
32             Container contentPane = getContentPane();
33             contentPane.setLayout(new BorderLayout());
34             dressingBoard = new DressingBoard();
35             contentPane.add("Center", dressingBoard);
36
37             garmentList = new JList();
38             garmentList.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
39             garmentList.addListSelectionListener(this);
40             contentPane.add("West", garmentList);
41
42             initList();
43         }
44         private void initList() {
45             String[] names =
46                 {
47                     "T-Shirt",
48                     "Briefs",
49                     "Left Sock",
50                     "Right Sock",
51                     "Shirt",
52                     "Pants",
53                     "Belt",
54                     "Tie",
55                     "Left Shoe",
56                     "Right Shoe" };
57             Point[] points =
58                 {
59                     new Point(75, 125),
60                     new Point(86, 197),
61                     new Point(127, 256),
62                     new Point(45, 260),
63                     new Point(69, 118),
64                     new Point(82, 199),
65                     new Point(88, 203),
66                     new Point(84, 124),
67                     new Point(129, 258),
68                     new Point(40, 268)};
69
70             Vector garments = new Vector(names.length);
71             for (int i = 0; i < names.length; ++i) {
72                 Image image =
73                     ResourceUtils.loadImage("chap14/images/" + names[i] + ".gif", this);
74                 Garment garment = new Garment(image, points[i].x, points[i].y, names[i]);
75                 garments.add(garment);
76             }

```

```

77     Collections.shuffle(garments);
78     garmentList.setListData(garments);
79 }
80 private void redrawBoy() {
81     ListModel lm = garmentList.getModel();
82     int stop = lm.getSize();
83     Vector order = new Vector();
84     for (int i = 0; i < stop; ++i) {
85         Garment garment = (Garment)lm.elementAt(i);
86         if (garment.isWorn()) {
87             order.add(garment);
88         }
89     }
90     dressingBoard.setOrder((Garment[])order.toArray(new Garment[0]));
91 }
92 public static void main(String[] arg) {
93     new MainFrame().setVisible(true);
94 }
95 public void valueChanged(ListSelectionEvent e) {
96     ListModel lm = garmentList.getModel();
97     for (int i = lm.getSize() - 1; i >= 0; --i) {
98         Garment garment = (Garment)lm.elementAt(i);
99         garment.setWorn(false);
100    }
101
102    Object[] selectedGarments = garmentList.getSelectedValues();
103    for (int i = 0; i < selectedGarments.length; ++i) {
104        Garment selectedGarment = (Garment)selectedGarments[i];
105        selectedGarment.setWorn(true);
106    }
107    redrawBoy();
108 }
109 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap14/gui1/MainFrame.java
java -cp classes chap14.gui1.MainFrame

```

Run the program and take time to play with it. Make single and multiple selections from the list and notice the effect each has on the drawing of the boy. Now comment out the toString() method of Garment, recompile and run the program again to observe the effect this has on the program. By default, JList uses a JLabel to render each list item and it sets the text of the JLabel to whatever toString() returns. We'll discuss renderers in the next section.

Quick Review

Swing's separable model architecture creates a loose coupling between a Component and the data it represents by defining interfaces with which the Component interacts, thus freeing an application to manage its data as it wishes. As long as the data is wrapped in the appropriate interface, the Component is automatically notified of changes in the data without any special effort on the part of the programmer.

STEP 3 — WRITING A CUSTOM RENDERER

Because this application will eventually support dragging, mouse clicks in the JList will be used for two purposes:

- Selecting a Garment item to toggle whether it is worn or not
- Selecting a Garment item to drag it to another location in the list

As you have already seen, the JList in the final version of this program is customized to display a checkbox next to each list item in order to distinguish these two use cases. A click in a checkbox toggles a Garment's worn property, while a click outside the checkbox is handled in the default manner for the JList. In this step we will learn how to customize the way that a JList displays its items.

USING A RENDERER

`JList` in the `gui1` package uses a `JLabel` to draw each item. How many `JLabels` do you think it has? One for each item? Ten? Actually, it has only one. A `JList` is an example of what I will call a *faux-composite* component. Other examples are `JTables`, `JTrees` and `JComboBoxes`. Faux-composite components can create the illusion of having thousands or more children components without incurring the overhead necessary to maintain that many components. Just like a painting of a door in a wall that looks real but isn't, what looks like a child component is only the image of a component. Faux-composite components use an object called a *renderer* to paint the component images or *cells*. A renderer implements a method that accepts several parameters detailing which cell is about to be painted (*the Object it represents, whether it's selected, etc.*) and returns a `Component` configured to represent that cell. When the faux-composite component needs to paint itself, it paints each visible cell one at a time by getting this `Component` from the renderer and painting its image at the cell's location. All the faux-composite components have default renderers but they also accept customized plug-in renderers.

`JList`, `JComboBox` and `JTree` use a single renderer. They each offer one method for getting this renderer and another for setting it. These methods are listed in table 14-7. The `JTable` class is more complex because it allows a different renderer to be set for each one of its columns and even for each column's header. Additionally it allows a different default renderer to be associated with each distinct column data class. The methods for getting and setting `JTable`'s and `TableColumn`'s renderers are listed in table 14-8.

Defining Class	Method
<code>JList</code>	<code>public void setCellRenderer(ListCellRenderer renderer)</code>
<code>JList</code>	<code>public ListCellRenderer getCellRenderer()</code>
<code>JTree</code>	<code>public void setCellRenderer(TreeCellRenderer renderer)</code>
<code>JTree</code>	<code>public TreeCellRenderer getCellRenderer()</code>
<code>JCombobox</code>	<code>public void setRenderer(ListCellRenderer renderer)</code>
<code>JCombobox</code>	<code>public ListCellRenderer getRenderer()</code>

Table 14-7: `JList`'s, `JTree`'s and `JComboBox`'s Renderer-Related Methods

Defining Class	Method
<code>JTable</code>	<code>public void setDefaultCellRenderer(Class columnClass, TableCellRenderer renderer)</code>
<code>JTable</code>	<code>public TableCellRenderer getDefaultCellRenderer(Class columnClass)</code>
<code>JTable</code>	<code>public TableCellRenderer getCellRenderer(int row, int column)</code>
<code>TableColumn</code>	<code>public void setCellRenderer (TableCellRenderer renderer)</code>
<code>TableColumn</code>	<code>public TableCellRenderer getCellRenderer()</code>
<code>TableColumn</code>	<code>public void setHeaderRenderer(TableCellRenderer renderer)</code>
<code>TableColumn</code>	<code>public TableCellRenderer getHeaderRenderer()</code>

Table 14-8: `JTable`'s Renderer-Related Methods

Figure 14-3 shows an example of a `JList` with a customized renderer. `JList` and `JComboBox` use `ListCellRenderers`. The `ListCellRenderer` interface defines the one method:

```
public Component getListCellRendererComponent( JList list,
                                               Object value, int index, boolean isSelected,
                                               boolean cellHasFocus)
```




Figure 14-3: A JFrame Containing a JList with a Custom ListCellRenderer

Figure 14-4 shows an example of a JTable using a customized renderer. JTables, TableColumn and JTableHeaders use TableCellRenderers. The TableCellRenderer interface defines the one method:

```
public Component getTableCellRendererComponent(
    JTable table,
    Object value,
    boolean isSelected,
    boolean hasFocus,
    int row,
    int column)
```

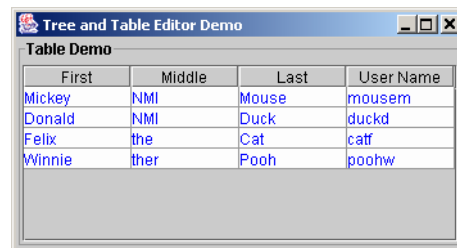


Figure 14-4: A JFrame Containing a JTable

Figure 14-5 shows a JTree with a highly customized renderer. JTrees use TreeCellRenderers. The TreeCellRenderer interface defines the one method:

```
public Component getTreeCellRendererComponent(
    JTree tree,
    Object value,
    boolean selected,
    boolean expanded,
    boolean leaf,
    int row,
    boolean hasFocus)
```

In each case, the first parameter is a reference to the faux-composite component itself. The second is the object (*obtained from the data model*) that is represented by the implicated cell. The remaining parameters identify the state and location of the implicated cell.

THE CHECKBOXLISTCELL CLASS

CheckboxListCell (*example 14.6*) will be the custom renderer for this application's JList. Because having a checkbox for a list cell might be a good idea in other contexts, we'll attempt to keep CheckboxListCell reusable by

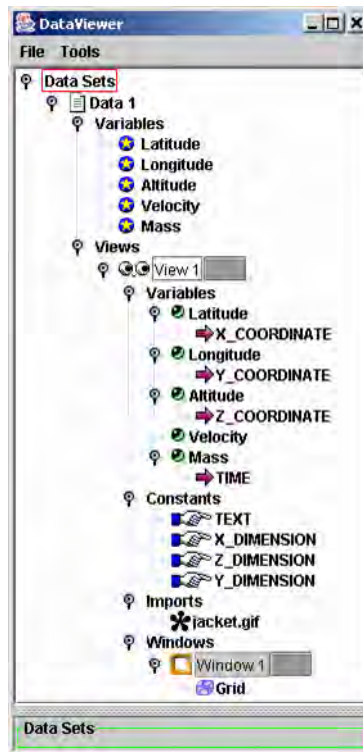


Figure 14-5: A JFrame Containing a Highly Customized JTree

avoiding any dependency on application-specific classes or data. If we're successful, we might be able to use it without modification in another program in the future.

14.6 chap14.gui2CheckboxListCell.java

```

1      package chap14.gui2;
2
3      import java.awt.BorderLayout;
4      import java.awt.Color;
5      import java.awt.Component;
6
7      import javax.swing.JCheckBox;
8      import javax.swing.JLabel;
9      import javax.swing.JList;
10     import javax.swing.JPanel;
11     import javax.swing.ListCellRenderer;
12
13     public abstract class CheckboxListCell implements ListCellRenderer {
14
15         private JLabel label;
16         private JCheckBox checkBox;
17         private JPanel panel;
18
19         public CheckboxListCell() {
20             panel = new JPanel(new BorderLayout());
21             label = new JLabel();
22             checkBox = new JCheckBox();
23             checkBox.setOpaque(false);
24             panel.add("Center", label);
25             panel.add("West", checkBox);
26         }
27
28         public Component getListCellRendererComponent(
29             JList list,
30             Object value,
31             int index,
32             boolean isSelected,
33             boolean cellHasFocus) {
34
35             panel.setBackground(isSelected ? Color.yellow : list.getBackground());
36             label.setText(String.valueOf(value));

```

```

37         checkBox.setSelected(getCheckedValue(value));
38         return panel;
39     }
40     protected abstract boolean getCheckedValue(Object value);
41 }

```

Because `CheckboxListCell` implements `ListCellRenderer`, its `getListCellRendererComponent()` method is obligated to return a `Component` that will represent the list item the list is currently attempting to paint. `CheckboxListCell.getListCellRendererComponent()` returns a panel (*created once at the time of construction*) containing a `JCheckbox` and a `JLabel`.

The `getListCellRendererComponent()` method takes advantage of its parameters to do just-in-time configuration of the renderer component. These parameters are a `JList` which is a reference to the `JList` itself, an `Object` which returns the same as invoking `getElementAt(index)` on its `ListModel` (*in our case the Object will be an instance of Garment*), an `int` which is the index of the object in the List, a `boolean` representing whether this list item is currently selected, and a `boolean` representing whether this list item currently has the focus (*for example, was it just clicked?*). We set the background of the renderer component to be yellow if it is selected, or the same as the `JList`'s background if it isn't selected. We set the `JLabel`'s text by calling `toString()` on the value parameter just as `JList`'s default renderer would.

We need to check or uncheck the checkbox depending on whether or not the garment is worn, but that presents a small dilemma. If we cast the value parameter to a `Garment` and call `Garment.isWorn()`, that would build in a dependency on the object being a `Garment`. We would like `CheckboxListCell` to be able to handle any object type that can be associated with the checked state of a checkbox. So, we need to figure out a way for it to determine the checked state without knowing the value parameter's type. There are different ways to approach this. The way we will employ here is probably the easiest to code. We make `CheckboxListCell` abstract, give it the abstract method `getCheckedValue()` thereby forcing subclasses to define the relationship by implementing this method.

By the way, did you notice that we needed to set the checkbox's opaque property to false? Some Swing components such as the `JLabel` have transparent backgrounds by default. These components only paint their foreground (*not their background*), thereby allowing any components behind them to show through the unpainted surface area. Other components have opaque backgrounds by default. These components take responsibility for painting their entire surface area. `JCheckbox` is one of these. If we were to let its opaque property default to true, the `JCheckbox` would paint its own background and might appear a different color than its container (*the JPanel*). I think it looks better with a transparent background.

THE MAINFRAME CLASS

The only change to `MainFrame` is that we plug in an instance of `CheckboxListCell` as the `JList`'s renderer (*lines 40 - 45 of example 14.7*).

14.7 chap14.gui2.MainFrame.java

```

1     package chap14.gui2;
2
3     import java.awt.BorderLayout;
4     import java.awt.Container;
5     import java.awt.Image;
6     import java.awt.Point;
7     import java.util.Collections;
8     import java.util.Vector;
9
10    import javax.swing.BorderFactory;
11    import javax.swing.JFrame;
12    import javax.swing.JList;
13    import javax.swing.ListModel;
14    import javax.swing.border.BevelBorder;
15    import javax.swing.event.ListSelectionEvent;
16    import javax.swing.event.ListSelectionListener;
17
18    import utils.ResourceUtils;
19    import chap14.gui0.DressingBoard;
20    import chap14.gui0.Garment;
21
22    public class MainFrame extends JFrame implements ListSelectionListener {
23
24        private DressingBoard dressingBoard;
25        private JList garmentList;
26

```

```

27     public JFrame() {
28         setTitle(getClass().getName());
29         setSize(600, 400);
30         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31
32         Container contentPane = getContentPane();
33         contentPane.setLayout(new BorderLayout());
34         dressingBoard = new DressingBoard();
35         contentPane.add("Center", dressingBoard);
36
37         garmentList = new JList();
38         garmentList.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
39         garmentList.addListSelectionListener(this);
40         CheckboxListCell cellHandler = new CheckboxListCell() {
41             protected boolean getCheckedValue(Object value) {
42                 return ((Garment)value).isWorn();
43             }
44         };
45         garmentList.setCellRenderer(cellHandler);
46         contentPane.add("West", garmentList);
47
48         initList();
49     }
50     private void initList() {
51         String[] names =
52             {
53                 "T-Shirt",
54                 "Briefs",
55                 "Left Sock",
56                 "Right Sock",
57                 "Shirt",
58                 "Pants",
59                 "Belt",
60                 "Tie",
61                 "Left Shoe",
62                 "Right Shoe" };
63         Point[] points =
64             {
65                 new Point(75, 125),
66                 new Point(86, 197),
67                 new Point(127, 256),
68                 new Point(45, 260),
69                 new Point(69, 118),
70                 new Point(82, 199),
71                 new Point(88, 203),
72                 new Point(84, 124),
73                 new Point(129, 258),
74                 new Point(40, 268)};
75
76         Vector garments = new Vector(names.length);
77         for (int i = 0; i < names.length; ++i) {
78             Image image =
79                 ResourceUtils.loadImage("chap14/images/" + names[i] + ".gif", this);
80             Garment garment = new Garment(image, points[i].x, points[i].y, names[i]);
81             garments.add(garment);
82         }
83         Collections.shuffle(garments);
84         garmentList.setListData(garments);
85     }
86     private void redrawBoy() {
87         ListModel lm = garmentList.getModel();
88         int stop = lm.getSize();
89         Vector order = new Vector();
90         for (int i = 0; i < stop; ++i) {
91             Garment garment = (Garment)lm.getElementAt(i);
92             if (garment.isWorn()) {
93                 order.add(garment);
94             }
95         }
96         dressingBoard.setOrder((Garment[])order.toArray(new Garment[0]));
97     }
98     public static void main(String[] arg) {
99         new JFrame().setVisible(true);
100     }
101     public void valueChanged(ListSelectionEvent e) {
102         ListModel lm = garmentList.getModel();
103         for (int i = lm.getSize() - 1; i >= 0; --i) {
104             Garment garment = (Garment)lm.getElementAt(i);
105             garment.setWorn(false);
106         }
107     }

```

```

108     Object[] selectedGarments = garmentList.getSelectedValues();
109     for (int i = 0; i < selectedGarments.length; ++i) {
110         Garment selectedGarment = (Garment)selectedGarments[i];
111         selectedGarment.setWorn(true);
112     }
113     redrawBoy();
114 }
115 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap14/gui2/MainFrame.java
java -cp classes chap14.gui2.MainFrame

```

Run this step’s program and notice the JList’s new looks! Notice also that the checkbox is not truly functional. Try selecting an unselected list item, for instance. Did your checkbox get checked? Hmm... Try to unselect that list item by selecting another. Did it become unchecked? Hmm... Try to uncheck a checked checkbox by clicking inside it. A checkbox should toggle state when it’s clicked, but this one doesn’t. Well, what did you expect? Remember that it’s actually only the image of a checkbox that our renderer painted there. But wait! If the checkbox is not functional (*it isn’t*) then how did it manage to display a check mark when its list item was selected and get unchecked when its list item was unselected? What’s really happening here? It is important that we be able to answer these questions so that we don’t lose control over a “maverick” program.

In words, here’s what happens when an unselected list cell is clicked: The click causes the cell to become selected. This causes the JList to call `MainFrame.valueChanged()` because `MainFrame` is the list’s registered `ListSelectionListener`. `MainFrame.valueChanged()` updates every `Garment`’s `worn` property to match the selection state and then calls `redrawBoy()`. Meanwhile, JList knows that it needs to repaint all cells whose selection state changed. When it is time to repaint, it calls `CheckboxListCell.getListCellRendererComponent()` for each cell that needs to be updated. `CheckboxListCell.getListCellRendererComponent()` sets the checkbox’s checked state to reflect whether or not the `Garment` is worn. Fortunately, the `Garment`’s worn state was just set by `MainFrame`’s `valueChanged()` method. Finally, the JList paints the affected cell with the correctly updated checkbox. Figure 14-6 details this process with a sequence diagram.

So, the reason unchecking a checked checkbox by clicking inside it doesn’t work, is that it doesn’t cause a change in the JList’s selection state. As far as the JList is concerned, there’s no need to repaint itself. Clearly, we’re going to need something more than a `ListSelectionListener` to get things functioning.

Quick Review

JList, JComboBox, JTree and JTable are examples of faux-composite components. They use renderers to create the illusion of having many children components without incurring the overhead of maintaining that many components. JList and JComboBox use `ListCellRenderers`; JTables use `TableCellRenderers`; and JTrees use `TreeCellRenderers`. By implementing the correct renderer interface, a custom renderer can affect a faux-composite component’s appearance, tailoring it to the data it represents.

STEP 4 —THE ART OF ILLUSION

Up until now, we have used a `ListSelectionEvent` to determine what garments the boy should wear. Now that we have plugged in our `CheckboxListCell` renderer complete with checkbox, we really need to make the checkbox functional and stop using a `ListSelectionEvent`. In this step, we improve the checkbox illusion by making the picture of the checkbox respond to `mouseClicks` as though it were real. We will also remove any code dealing with `ListSelectionEvents` and `ListSelectionListeners` since our “checkbox” is going to be responsible for the toggling of the `Garment`’s `worn` property.

If this were a normal checkbox, we could just give it an `ActionListener` that calls `Garment.isWorn(JCheckbox.isSelected())`. But this is not a real checkbox. When the user clicks on what he thinks is a checkbox, he is really only clicking a picture of a checkbox inside the JList. In other words, he is clicking in the JList itself. So we need to add a `MouseListener` to the JList and implement the `mouseClicked()` method. Theoretically speaking, if we can figure

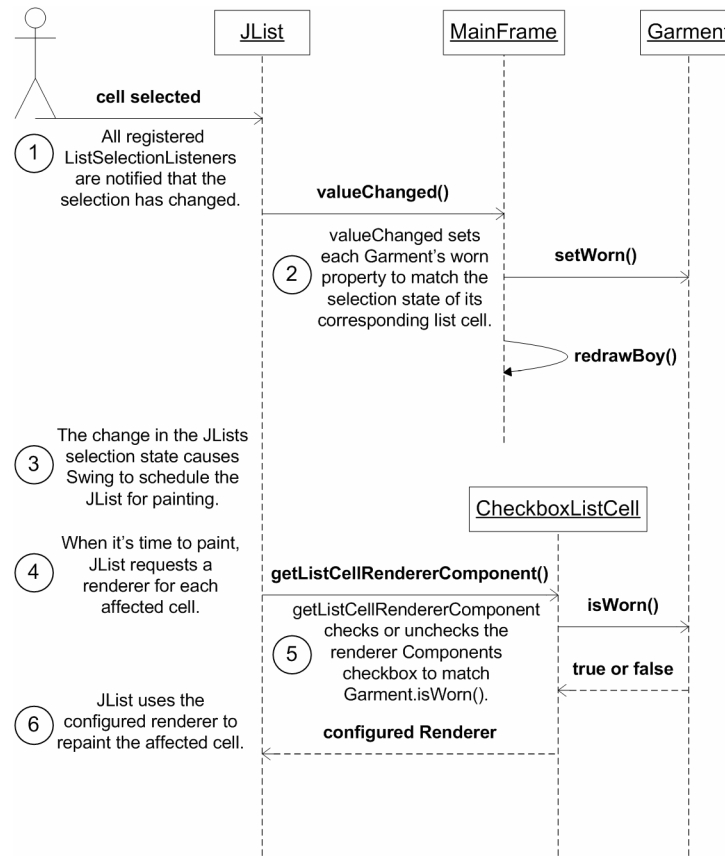


Figure 14-6: Sequence Diagram for a JList Using CheckboxListCell – First Version

out whether or not the click was inside the checkbox, we'll know whether or not we should toggle the Garment's worn property.

Here is our mission: Starting with the JList, the CheckboxListCell renderer component and the location of the mouse click, navigate our way through the Swing API to determine whether or not the click was inside the imaginary checkbox. One of the marks of a good API is the availability of methods that support even unanticipated uses such as this one. Swing is a good API. Of course, it's also a big API and there are lots and lots of methods and classes to look through when you're trying to get from point A to point B. Here is the path I found:

Pass the location of the mouseclick into `JList.locationToIndex()` to get the index of the list cell that contained the mouseclick. Pass this index into `JList.getCellBounds()` to get the bounds of that list cell. Set the bounds of the renderer component to these bounds. (*This is similar to what JList might do preparatory to rendering this list cell*). From the resized renderer component, get the bounds of its checkbox. Remember that a component's bounds are in its parent's coordinate system, so this checkbox's bounds are in the coordinate system of the renderer component. Now, pass the JList, the location of the mouseclick and the renderer component into `SwingUtilities.convertPoint()` to convert the mouseclick location from the JList's coordinate system into the renderer component's coordinate system. This converted point and the checkbox's bounds are now in the same coordinate system. Pass the converted point into the `contains()` method of the checkbox's bounds to find out if the mouseclick was inside the checkbox. Figure 14-7 illustrates this path graphically.

Whew! That's a bunch of maneuvering through the Swing API! Anyway, once we determine that the click is in the checkbox, we can handle it however we want. Whatever class we register as a `MouseListener` to the JList will have access to the JList and the `mouseClick` location, since the JList will be the source of the event and the location will be an attribute of the `MouseEvent`. However, not every class knows how to get to the Checkbox. This makes `CheckboxListCell` the logical place to put this code for two reasons: 1) `CheckboxListCell` constructs the renderer component, so it has access to the checkbox, and 2) Keeping the checkbox-finding logic inside `CheckboxListCell`

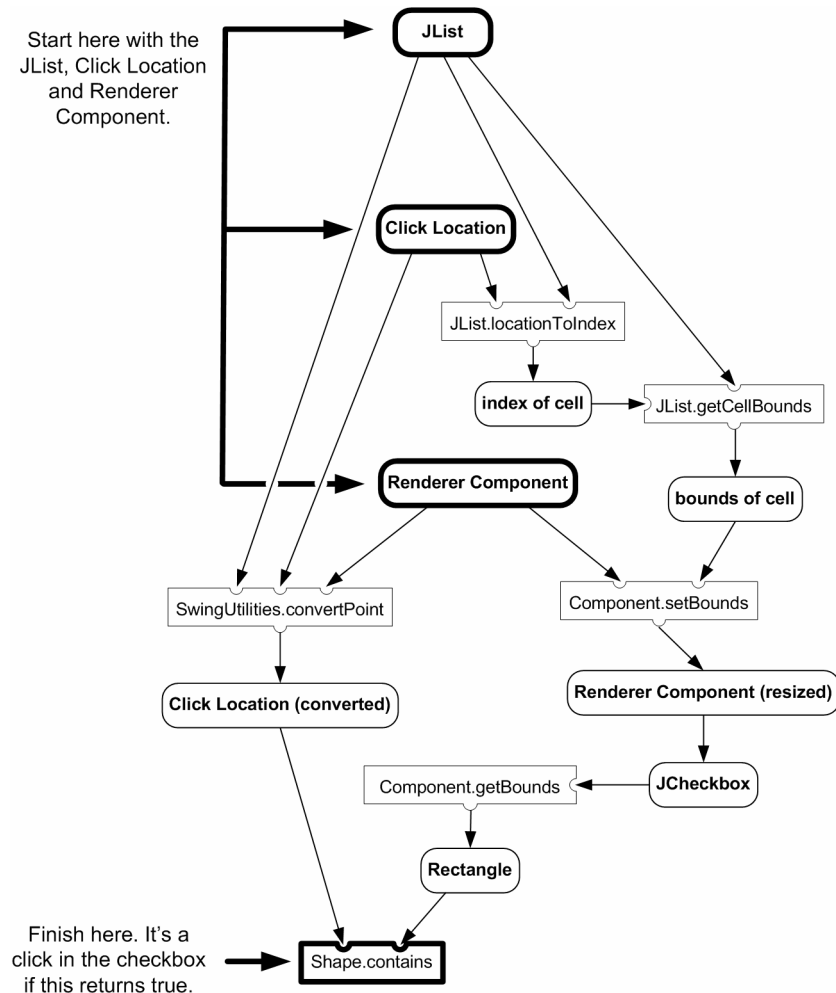


Figure 14-7: Maneuvering Through the Swing API

minimizes the chance that future modifications to the structure of `CheckboxListCell`'s renderer component could break external dependent classes.

So, we'll have `CheckboxListCell` implement `MouseListener` and we'll register it with the `JList`. For now, even though it introduces a dependency on the `Garment` class, we will put the code to toggle the clicked `Garment`'s worn property right inside `CheckboxListCell`'s `mouseClicked` method. We'll remove this dependency in the next step. We're not done yet, though. Since the checkbox isn't real, it is our responsibility to update its appearance when it's been toggled. But that's simple. All we need to do is to tell the `JList` to repaint itself. When it paints this particular cell, the `JList` will obtain the `CheckboxListCell` renderer which will have already set the selected state of its checkbox to match whether or not the `Garment` is worn. Since we only need to repaint one list cell, we tell the `JList` to repaint only the rectangular area occupied by that list cell (*line 77 of example 14.8*).

Now that we know how to determine whether or not a mouseclick is in the checkbox, writing the code isn't difficult. The `mouseClicked` method does exactly what we have laid out with one addition. I have never liked it that a click in the empty area below the last list cell of a `JList` acts like a click in the last cell. I think it would be better used as a method of unselecting all cells. So, if the mouseclick is not in the selected cell and the shift-key and control-keys weren't down, lines 57 - 62 clear the selection and return.

Example 14.8 gives the code for the modified version of the `CheckboxListCell` class.

THE CHECKBOXLISTCELL CLASS

14.8 chap14.gui3.CheckboxListCell.java

```

1      package chap14.gui3;
2
3      import java.awt.BorderLayout;
4      import java.awt.Color;
5      import java.awt.Component;
6      import java.awt.Point;
7      import java.awt.Rectangle;
8      import java.awt.event.MouseEvent;
9      import java.awt.event.MouseListener;
10
11     import javax.swing.JCheckBox;
12     import javax.swing.JLabel;
13     import javax.swing.JList;
14     import javax.swing.JPanel;
15     import javax.swing.ListCellRenderer;
16     import javax.swing.ListModel;
17     import javax.swing.SwingUtilities;
18
19     import chap14.gui0.Garment;
20
21     public abstract class CheckboxListCell
22     implements ListCellRenderer, MouseListener {
23
24     private JLabel label;
25     private JCheckBox checkBox;
26     private JPanel panel;
27
28     public CheckboxListCell() {
29         panel = new JPanel(new BorderLayout());
30         label = new JLabel();
31         checkBox = new JCheckBox();
32         checkBox.setOpaque(false);
33         panel.add("Center", label);
34         panel.add("West", checkBox);
35     }
36
37     public Component getListCellRendererComponent(
38         JList list,
39         Object value,
40         int index,
41         boolean isSelected,
42         boolean cellHasFocus) {
43
44         panel.setBackground(isSelected ? Color.yellow : list.getBackground());
45         label.setText(String.valueOf(value));
46         checkBox.setSelected(getCheckedValue(value));
47         return panel;
48     }
49     protected abstract boolean getCheckedValue(Object value);
50
51     public void mouseClicked(MouseEvent e) {
52         JList list = (JList)e.getSource();
53         Point p = e.getPoint();
54         int index = list.locationToIndex(p);
55         Rectangle rect = list.getCellBounds(index, index);
56         panel.setBounds(rect);
57         if (!rect.contains(p)) {
58             if (!e.isShiftDown() && !e.isControlDown()) {
59                 list.clearSelection();
60             }
61             return;
62         }
63         p = SwingUtilities.convertPoint(list, p, panel);
64         if (!checkBox.getBounds().contains(p)) {
65             return;
66         }
67
68         /******
69         /* Not a good design. CheckboxCellHandler should be flexible.      */
70         /* We will remove this dependence on Garment in the next iteration. */
71         /******
72         ListModel model = list.getModel();
73         Garment garment = (Garment)model.getElementAt(index);
74         garment.setWorn(!garment.isWorn());
75         /******
76
77         list.repaint(rect.x, rect.y, rect.width, rect.height);

```



```

78     }
79     public void mousePressed(MouseEvent e) {}
80     public void mouseReleased(MouseEvent e) {}
81     public void mouseEntered(MouseEvent e) {}
82     public void mouseExited(MouseEvent e) {}
83 }

```

THE MAINFRAME CLASS

We only need to make minor changes to `MainFrame`. We “unimplement” `ListSelectionListener` and remove the `valueChanged(ListSelectionEvent)` method. Also, instead of registering `MainFrame` as a `ListSelectionListener` with the `JList`, we register `CheckboxListCell` as a `MouseListener` to the `JList` in line 42.

14.9 chap14.gui3.MainFrame.java

```

1     package chap14.gui3;
2
3     import java.awt.BorderLayout;
4     import java.awt.Container;
5     import java.awt.Image;
6     import java.awt.Point;
7     import java.util.Collections;
8     import java.util.Vector;
9
10    import javax.swing.BorderFactory;
11    import javax.swing.JFrame;
12    import javax.swing.JList;
13    import javax.swing.ListModel;
14    import javax.swing.border.BevelBorder;
15
16    import utils.ResourceUtils;
17    import chap14.gui0.DressingBoard;
18    import chap14.gui0.Garment;
19
20    public class MainFrame extends JFrame {
21
22        private DressingBoard dressingBoard;
23        private JList garmentList;
24
25        public MainFrame() {
26            setTitle(getClass().getName());
27            setSize(600, 400);
28            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29
30            Container contentPane = getContentPane();
31            contentPane.setLayout(new BorderLayout());
32            dressingBoard = new DressingBoard();
33            contentPane.add("Center", dressingBoard);
34
35            garmentList = new JList();
36            garmentList.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
37            CheckboxListCell cellHandler = new CheckboxListCell() {
38                protected boolean getCheckedValue(Object value) {
39                    return ((Garment)value).isWorn();
40                }
41            };
42            garmentList.addMouseListener(cellHandler);
43            garmentList.setCellRenderer(cellHandler);
44            contentPane.add("West", garmentList);
45
46            initList();
47        }
48        private void initList() {
49            String[] names =
50                {
51                    "T-Shirt",
52                    "Briefs",
53                    "Left Sock",
54                    "Right Sock",
55                    "Shirt",
56                    "Pants",
57                    "Belt",
58                    "Tie",
59                    "Left Shoe",
60                    "Right Shoe" };
61            Point[] points =
62                {
63                    new Point(75, 125),
64                    new Point(86, 197),

```

```

65         new Point(127, 256),
66         new Point(45, 260),
67         new Point(69, 118),
68         new Point(82, 199),
69         new Point(88, 203),
70         new Point(84, 124),
71         new Point(129, 258),
72         new Point(40, 268)};
73
74     Vector garments = new Vector(names.length);
75     for (int i = 0; i < names.length; ++i) {
76         Image image =
77             ResourceUtils.loadImage("chap14/images/" + names[i] + ".gif", this);
78         Garment garment = new Garment(image, points[i].x, points[i].y, names[i]);
79         garments.add(garment);
80     }
81     Collections.shuffle(garments);
82     garmentList.setListData(garments);
83 }
84 private void redrawBoy() {
85     ListModel lm = garmentList.getModel();
86     int stop = lm.getSize();
87     Vector order = new Vector();
88     for (int i = 0; i < stop; ++i) {
89         Garment garment = (Garment)lm.getElementAt(i);
90         if (garment.isWorn()) {
91             order.add(garment);
92         }
93     }
94     dressingBoard.setOrder((Garment[])order.toArray(new Garment[0]));
95 }
96 public static void main(String[] arg) {
97     new MainFrame().setVisible(true);
98 }
99 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap14/gui3/MainFrame.java
java -cp classes chap14.gui3.MainFrame

```

Run the program. Great! Our checkboxes work. Unfortunately, they don't affect the image in DressingBoard because we forgot to call `redrawBoy`. The previous step called `redrawBoy()` from `MainFrame.valueChanged()`, but when we unimplemented `ListSelectionListener`, we removed the `valueChanged()` method and the only call to `redrawBoy()`. Oh well, we've done enough for this step. In the next step, we'll get rid of `CheckboxListCell`'s dependence on `Garment` and we'll make sure to redraw the boy.

Quick Review

The Swing API can be overwhelming because of the sheer number of classes and methods to come to terms with. But, as this step illustrated, it has the flexibility to support even non-standard uses.

STEP 5 – DEFINING YOUR OWN EVENTS AND EVENTLISTENERS

In this step, we will create our own event and event listener types and write a mechanism similar to Swing's mechanism for registering event listeners with components. In the process we will gain better insight into the purpose and value of Swing's event/listener mechanism.

THE CHECKBOXLISTCELL CLASS

DressingBoard used to update as a response to a `ListSelectionEvent` but now it has to respond to a mouse click inside an imposter checkbox. Somehow, `MainFrame` needs to be notified when `CheckboxListCell`'s "checkbox" has been toggled just as it used to be notified of `ListSelectionEvents`. Since only `CheckboxListCell` knows when the "checkbox" has been toggled, it is the logical candidate for handling notification. We will create the new event type, `ToggleEvent`, to define the event of a click inside the checkbox, and we will define the new listener interface, `Tog-`

gleListener, such that Objects that implement ToggleListener and are registered with CheckboxListCell will be notified whenever a ToggleEvent occurs. In this way, our code design will loosely mirror Swing’s own event/listener mechanism.

CheckboxListCell will generate a ToggleEvent when it determines that the user has clicked inside its checkbox image. Two bits of information that CheckboxListCell will have at this point are the JList reference and the index of the cell whose checkbox was toggled. We will package these into the ToggleEvent object and create a ToggleEvent constructor that requires them. The JList will be the source of the event and the index might prove to be useful to a ToggleListener. The ToggleEvent class is defined in lines 76 - 86 of example 14.10.

Because ToggleEvent and ToggleListener exist solely to support CheckboxListCell’s special needs, we define them as inner classes of CheckboxListCell. They could be defined as external classes but that would suggest that they had more universal utility. ToggleEvent is declared as a static class so that the existence of a ToggleEvent instance will not depend on the existence of an instance of its enclosing CheckboxListCell class. This would allow an external class to create a new instance of a ToggleEvent by the statement:

```
CheckboxListCell.ToggleEvent event = new CheckboxListCell.ToggleEvent(list,
index);
```

If it were not a static class, then in order for an external class to create a new instance of a ToggleEvent there would already have to be an instance of CheckboxListCell from which to create it as in:

```
CheckboxListCell cell = new CheckboxListCell();
CheckboxListCell.ToggleEvent event = cell.new ToggleEvent(list, index);
```

This is a moot point for this application’s purposes since the only class that will be creating ToggleEvents is CheckboxListCell itself. Because CheckboxListCell is the enclosing class, it can use the simpler notation:

```
ToggleEvent event = new ToggleEvent(list, index);
```

The ToggleListener interface is defined in lines 87 - 89 of example 14.10. It declares only one method: checkboxToggled(ToggleEvent). Interfaces are inherently “static” as they are not really objects at all. So an external class could create a new instance of ToggleListener like so:

```
CheckboxListCell.ToggleListener tl = new CheckboxListCell.ToggleListener () {
    public void checkboxToggled (ToggleEvent event) {}
};
```

We will not, however, create one this way. Instead, we will simply have the existing class, MainFrame, implement ToggleListener.

In order for CheckboxListCell to maintain registered ToggleListeners, we copy the pattern set by many Swing classes before us and add the methods addToggleListener(ToggleListener) and removeToggleListener(ToggleListener). These methods update a Vector that contains all registered ToggleListeners. In lines 90 - 95, a third method called “notifyListeners()” handles listener notification by passing a ToggleEvent to the checkboxToggled method of all registered ToggleListeners. With this event-listener mechanism in place, we can remove the dependency on Garment and leave it to each particular ToggleListener to respond, as needed, to the ToggleEvent.

14.10 chap14.gui4.CheckboxListCell.java

```
1     package chap14.gui4;
2
3     import java.awt.BorderLayout;
4     import java.awt.Color;
5     import java.awt.Component;
6     import java.awt.Point;
7     import java.awt.Rectangle;
8     import java.awt.event.MouseEvent;
9     import java.awt.event.MouseListener;
10    import java.util.EventListener;
11    import java.util.EventObject;
```

```

12     import java.util.Iterator;
13     import java.util.Vector;
14
15     import javax.swing.JCheckBox;
16     import javax.swing.JLabel;
17     import javax.swing.JList;
18     import javax.swing.JPanel;
19     import javax.swing.ListCellRenderer;
20     import javax.swing.SwingUtilities;
21
22     public abstract class CheckboxListCell
23     implements ListCellRenderer, MouseListener {
24
25         private JLabel label;
26         private JCheckBox checkBox;
27         private JPanel panel;
28         private Vector listeners = new Vector();
29
30         public CheckboxListCell() {
31             panel = new JPanel(new BorderLayout());
32             label = new JLabel();
33             checkBox = new JCheckBox();
34             checkBox.setOpaque(false);
35             panel.add("Center", label);
36             panel.add("West", checkBox);
37         }
38         public Component getListCellRendererComponent(
39             JList list,
40             Object value,
41             int index,
42             boolean isSelected,
43             boolean cellHasFocus) {
44
45             panel.setBackground(isSelected ? Color.yellow : list.getBackground());
46             label.setText(String.valueOf(value));
47             checkBox.setSelected(getCheckedValue(value));
48             return panel;
49         }
50         protected abstract boolean getCheckedValue(Object value);
51
52         public void mouseClicked(MouseEvent e) {
53             JList list = (JList)e.getSource();
54             Point p = e.getPoint();
55             int index = list.locationToIndex(p);
56             Rectangle rect = list.getCellBounds(index, index);
57             panel.setBounds(rect);
58             if (!rect.contains(p)) {
59                 if (!e.isShiftDown() && !e.isControlDown()) {
60                     list.clearSelection();
61                 }
62                 return;
63             }
64             p = SwingUtilities.convertPoint(list, p, panel);
65             if (!checkBox.getBounds().contains(p)) {
66                 return;
67             }
68             notifyListeners(new ToggleEvent(list, index));
69             list.repaint(rect.x, rect.y, rect.width, rect.height);
70         }
71         public void mousePressed(MouseEvent e) {}
72         public void mouseReleased(MouseEvent e) {}
73         public void mouseEntered(MouseEvent e) {}
74         public void mouseExited(MouseEvent e) {}
75
76         public static class ToggleEvent extends EventObject {
77             private int index;
78
79             public ToggleEvent(JList list, int index) {
80                 super(list);
81                 this.index = index;
82             }
83             public int getIndex() {
84                 return index;
85             }
86         }
87         public interface ToggleListener extends EventListener {
88             public void checkBoxToggled(ToggleEvent event);
89         }
90         private void notifyListeners(ToggleEvent event) {
91             for (Iterator it = listeners.iterator(); it.hasNext(); ) {
92                 ToggleListener rl = (ToggleListener)it.next();

```

```

93         rl.checkboxToggled(event);
94     }
95 }
96 public void addToggleListener(ToggleListener rl) {
97     listeners.add(rl);
98 }
99 public void removeToggleListener(ToggleListener rl) {
100     listeners.remove(rl);
101 }
102 }

```

THE MAINFRAME CLASS

The changes to `MainFrame` are minor. It implements `CheckboxListCell.ToggleListener()`, defines the `checkboxToggled()` method (*lines 100 - 106*) and registers itself as a `ToggleListener` with the instance of `CheckboxListCell` (*line 45*). Just as its predecessor two steps ago called `redrawBoy()` whenever it was notified of a `ListSelectionEvent`, this version of `MainFrame` will call `redrawBoy()` whenever it is notified of a `ToggleEvent`. But first, its `checkboxToggled()` method will use the `ToggleEvent`'s index to determine which `Garment` was affected and toggle that `Garment`'s worn property (*lines 102 - 104*).

14.11 *chap14.gui4.MainFrame.java*

```

1     package chap14.gui4;
2
3     import java.awt.BorderLayout;
4     import java.awt.Container;
5     import java.awt.Image;
6     import java.awt.Point;
7     import java.util.Collections;
8     import java.util.Vector;
9
10    import javax.swing.BorderFactory;
11    import javax.swing.JFrame;
12    import javax.swing.JList;
13    import javax.swing.ListModel;
14    import javax.swing.border.BevelBorder;
15
16    import chap14.gui0.DressingBoard;
17    import chap14.gui0.Garment;
18    import utils.ResourceUtils;
19    import chap14.gui4.CheckboxListCell;
20
21    public class MainFrame
22        extends JFrame
23        implements CheckboxListCell.ToggleListener {
24
25        private DressingBoard dressingBoard;
26        private JList garmentList;
27
28        public MainFrame() {
29            setTitle(getClass().getName());
30            setSize(600, 400);
31            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
32
33            Container contentPane = getContentPane();
34            contentPane.setLayout(new BorderLayout());
35            dressingBoard = new DressingBoard();
36            contentPane.add("Center", dressingBoard);
37
38            garmentList = new JList();
39            garmentList.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
40            CheckboxListCell cellHandler = new CheckboxListCell() {
41                protected boolean getCheckedValue(Object value) {
42                    return ((Garment)value).isWorn();
43                }
44            };
45            cellHandler.addToggleListener(this);
46            garmentList.addMouseListener(cellHandler);
47            garmentList.setCellRenderer(cellHandler);
48            contentPane.add("West", garmentList);
49
50            initList();
51        }
52        private void initList() {
53            String[] names =
54                {
55                "T-Shirt",
56                "Briefs",

```

```

57         "Left Sock",
58         "Right Sock",
59         "Shirt",
60         "Pants",
61         "Belt",
62         "Tie",
63         "Left Shoe",
64         "Right Shoe" };
65     Point[] points =
66     {
67         new Point(75, 125),
68         new Point(86, 197),
69         new Point(127, 256),
70         new Point(45, 260),
71         new Point(69, 118),
72         new Point(82, 199),
73         new Point(88, 203),
74         new Point(84, 124),
75         new Point(129, 258),
76         new Point(40, 268)};
77
78     Vector garments = new Vector(names.length);
79     for (int i = 0; i < names.length; ++i) {
80         Image image =
81             ResourceUtils.loadImage("chap14/images/" + names[i] + ".gif", this);
82         Garment garment = new Garment(image, points[i].x, points[i].y, names[i]);
83         garments.add(garment);
84     }
85     Collections.shuffle(garments);
86     garmentList.setListData(garments);
87 }
88 private void redrawBoy() {
89     ListModel lm = garmentList.getModel();
90     int stop = lm.getSize();
91     Vector order = new Vector();
92     for (int i = 0; i < stop; ++i) {
93         Garment garment = (Garment)lm.getElementAt(i);
94         if (garment.isWorn()) {
95             order.add(garment);
96         }
97     }
98     dressingBoard.setOrder((Garment[])order.toArray(new Garment[0]));
99 }
100 public void checkboxToggled(CheckboxListCell.ToggleEvent event) {
101     JList list = (JList)event.getSource();
102     int index = event.getIndex();
103     Garment garment = (Garment)list.getModel().getElementAt(index);
104     garment.setWorn(!garment.isWorn());
105     redrawBoy();
106 }
107 public static void main(String[] arg) {
108     new MainFrame().setVisible(true);
109 }
110 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap14/gui4/MainFrame.java
java -cp classes chap14.gui4.MainFrame

```

Run the program. Success! Checkboxes now affect the drawing of the boy.

Here's what happens when we click in the checkbox of one of the JList's cells: The click in the checkbox causes the JList to call `CheckboxListCell.mouseClicked()` because `CheckboxListCell` is a registered `MouseListener` of the list. `CheckboxListCell.mouseClicked()` determines that the click was in the checkbox and calls `MainFrame.checkboxToggled()` because `MainFrame` is a registered `ToggleListener` of `CheckboxListCell`. `MainFrame.checkboxToggled()` toggles the affected `Garment`'s worn property and then calls `redrawBoy()`.

Meanwhile, `CheckboxListCell.mouseClicked()` calls `repaint()` for the affected cell which causes Swing to schedule the JList for repainting. When it is time to repaint the affected cells, JList calls `CheckboxListCell.getListCellRendererComponent()` for the affected cell, and `CheckboxListCell.getListCellRendererComponent()` sets the checkbox's checked state to reflect whether or not the `Garment` is worn. Fortunately, the `Garment`'s worn state was just set by `MainFrame`'s `checkboxToggled()` method. Finally, the JList paints the affected cell with the correctly updated checkbox. Figure 14-8 details this process in a sequence diagram.

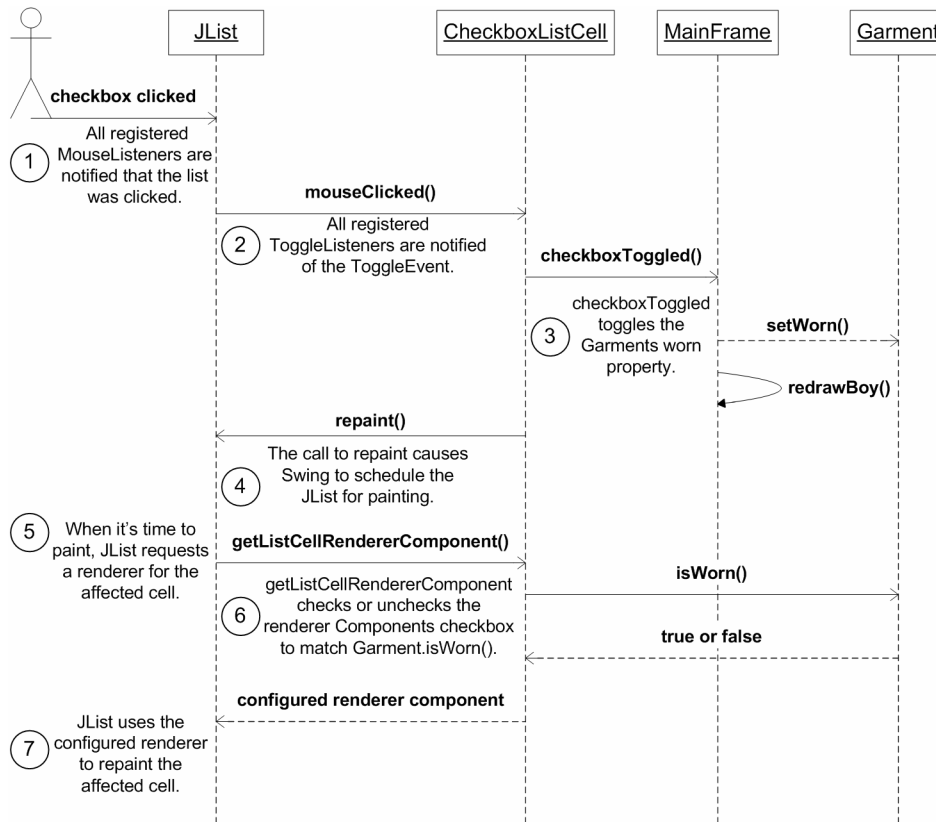


Figure 14-8: Sequence Diagram for a JList Using CheckboxListCell – Second Version

Quick Review

In this step, we created our own event and event listener types and wrote a mechanism similar to Swing’s mechanism for registering EventListeners with components.

INTERLUDE – WRITING A CUSTOM EDITOR

Looking back over the previous two steps (*steps 4 and 5*), what we did could be summed up as “making an editable list cell”. The renderer gave us the image of a checkbox but it took some effort to coax life into that image. Earlier it was pointed out that JTrees, JTables and JComboBoxes use custom renderers just as JList does. What wasn’t mentioned was that all of them except for the JList are editable and offer a mechanism for customizing editors that is similar to the mechanism for customizing renderers, and that greatly simplifies editing their underlying data models. This chapter’s main program doesn’t require an editable JTable, JTree or JComboBox, so in this brief interlude, we take a quick detour to look at the process for customizing JTree and JTable editors.

JTrees use TreeCellEditors. Table 14-9 shows JTree’s editor-related methods. JTables and TableColumn use TableCellEditors. Table 14-10 shows JTable’s and TableColumn’s editor-related methods.

Method Name
<code>public void setCellEditor(TreeCellEditor editor)</code>
<code>public TreeCellEditor getCellEditor()</code>

Table 14-9: JTree's Editor-Related Methods

Defining Class	Method
JTable	<code>public void setDefaultEditor(Class columnClass, TableCellEditor editor)</code>
JTable	<code>public TableCellRenderer getDefaultEditor(Class columnClass)</code>
JTable	<code>public TableCellRenderer getCellEditor(int row, int column)</code>
TableColumn	<code>public void setHeaderEditor(TableCellEditor headerEditor)</code>
TableColumn	<code>public TableCellEditor getHeaderEditor()</code>
TableColumn	<code>public void setCellEditor(TableCellEditor cellEditor)</code>
TableColumn	<code>public TableCellRenderer getCellEditor()</code>

Table 14-10: JTable's and TableColumn's Editor-Related Methods

Both the `TreeCellEditor` and `TableCellEditor` interfaces extend from the `CellEditor` interface as shown in the class hierarchy in figure 14-9. Table 14-11 shows the methods that the `CellEditor` interface defines.

```

javax.swing.CellEditor
  javax.swing.tree.TreeCellEditor
  javax.swing.table.TableCellEditor

```

Figure 14-9: TreeCellEditor and TableCellEditor Inheritance Hierarchy

Method Name and Purpose
public Object getCellEditorValue() Returns the value that the editor component contains.
public boolean isCellEditable(EventObject anEvent) Returns true if the cell should be edited.
public boolean shouldSelectCell(EventObject anEvent) Returns true if the cell should be selected.
public boolean stopCellEditing() Call this to stop the editing process and accept the value contained in the editor component. This method returns false if the value could not be accepted.
public void cancelCellEditing() Call this to stop the editing process and discard the value contained in the editor component.

Table 14-11: javax.swing.CellEditor Methods

public void addCellEditorListener(CellEditorListener l) Called automatically when this CellEditor becomes a component's editor.
public void removeCellEditorListener(CellEditorListener l) Called automatically when this CellEditor is no longer a component's editor.

Table 14-11: javax.swing.CellEditor Methods

The TableCellEditor interface extends CellEditor and defines one additional method:

```
public Component getTableCellEditorComponent( JTable table,
                                             Object value,
                                             boolean isSelected,
                                             int row,
                                             int column);
```

The TreeCellEditor interface extends CellEditor and defines the additional method:

```
public Component getTreeCellEditorComponent( JTree tree,
                                             Object value,
                                             boolean isSelected,
                                             boolean expanded,
                                             boolean leaf,
                                             int row);
```

Both these methods must return a configured component that will handle the editing of the value parameter. In both cases, the first parameter is a reference to the faux-composite component itself. The second is the object (*obtained from the data model*) that is represented by the implicated cell. The remaining parameters identify the state and location of the implicated cell.

As an aid to writing a CellEditor, the javax.swing.AbstractCellEditor class implements CellEditor to provide default implementations of all the methods except for getCellEditorValue. Whether you need an editor for a JTable or a JTree, my recommendation is to extend AbstractCellEditor and implement either the TreeCellEditor or TableCellEditor interface as required. You will have to implement two methods: getCellEditorValue() and the one method declared by the additional interface. You may choose to override other methods if their default behavior is not appropriate. In particular, you might want to override the two methods isCellEditable() and shouldSelectCell() to return different values depending on which element of the data model is implicated by the EventObject. Determining which element is implicated by the EventObject takes a bit of work, though, as the getValueFromEvent() method of DemoTreeOrTableHandler class illustrates (*lines 90 - 108 of example 14.12*).

Following is a demonstration application that creates a JTree and a JTable and a minimal renderer and editor for each. It is comprised of the two classes DemoTreeOrTableCellHandler and DemoFrame.

THE DEMOTREEORTABLECELLHANDLER CLASS

The DemoTreeOrTableCellHandler class extends AbstractCellEditor. In an attempt to “be all things to all people” it implements TreeCellEditor, TableCellEditor, TreeCellRenderer and TableCellRenderer simultaneously, albeit in a minimal way. This is perhaps putting too much functionality into one class but it is useful for pointing out the overlaps between the various interfaces.

14.12 chap14.interlude.DemoTreeOrTableCellHandler.java

```
1      package chap14.interlude;
2
3      import java.awt.Color;
4      import java.awt.Component;
5      import java.awt.Point;
6      import java.awt.event.ActionEvent;
7      import java.awt.event.ActionListener;
8      import java.awt.event.MouseEvent;
9      import java.util.EventObject;
10
11     import javax.swing.AbstractCellEditor;
```

```

12     import javax.swing.JTable;
13     import javax.swing.JTextField;
14     import javax.swing.JTree;
15     import javax.swing.table.TableCellEditor;
16     import javax.swing.table.TableCellRenderer;
17     import javax.swing.tree.TreeCellEditor;
18     import javax.swing.tree.TreeCellRenderer;
19     import javax.swing.tree.TreePath;
20
21     public class DemoTreeOrTableCellHandler
22     extends AbstractCellEditor
23     implements TreeCellEditor, TableCellEditor, TreeCellRenderer, TableCellRenderer {
24
25     private JTextField renderField;
26     private JTextField editorField;
27
28     public DemoTreeOrTableCellHandler() {
29         renderField = new JTextField();
30         renderField.setBorder(null);
31         renderField.setForeground(Color.blue);
32
33         editorField = new JTextField();
34         editorField.setBorder(null);
35         editorField.setForeground(Color.red);
36
37         editorField.addActionListener(new ActionListener() {
38             public void actionPerformed(ActionEvent e) {
39                 stopCellEditing();
40             }
41         });
42     }
43     public Object getCellEditorValue() {
44         return editorField.getText();
45     }
46     public Component getTreeCellEditorComponent(
47         JTree tree,
48         Object value,
49         boolean isSelected,
50         boolean expanded,
51         boolean leaf,
52         int row) {
53
54         editorField.setText(String.valueOf(value));
55         return editorField;
56     }
57     public Component getTableCellEditorComponent(
58         JTable table,
59         Object value,
60         boolean isSelected,
61         int row,
62         int column) {
63
64         editorField.setText(String.valueOf(value));
65         return editorField;
66     }
67     public Component getTreeCellRendererComponent(
68         JTree tree,
69         Object value,
70         boolean selected,
71         boolean expanded,
72         boolean leaf,
73         int row,
74         boolean hasFocus) {
75
76         renderField.setText(String.valueOf(value));
77         return renderField;
78     }
79     public Component getTableCellRendererComponent(
80         JTable table,
81         Object value,
82         boolean isSelected,
83         boolean hasFocus,
84         int row,
85         int column) {
86
87         renderField.setText(String.valueOf(value));
88         return renderField;
89     }
90     private Object getValueFromEvent(EventObject e) {
91         Object value = null;
92         if (e instanceof MouseEvent) {

```

```

93         MouseEvent event = (MouseEvent)e;
94         Point p = event.getPoint();
95         Object src = e.getSource();
96         if (src instanceof JTree) {
97             JTree tree = (JTree)src;
98             TreePath path = tree.getClosestPathForLocation(p.x, p.y);
99             value = path.getLastPathComponent();
100        } else if (src instanceof JTable) {
101            JTable table = (JTable)src;
102            int row = table.rowAtPoint(p);
103            int column = table.columnAtPoint(p);
104            value = table.getValueAt(row, column);
105        }
106    }
107    return value;
108 }
109 }

```

THE DEMOFRAME CLASS

The DemoFrame class illustrates the basics of creating a TableModel and TreeModel as well as demonstrating the use of a custom renderer and editor in trees and tables. Notice that it creates two instances of DemoTreeOrTableHandler – one for the JTree and another for the JTable. Editors should not be shared between components lest the components’ painting processes become confused. For some mischievous fun, rewrite DemoFrame so that the JTree and JTable share the same instance of DemoTreeOrTableHandler. Then run the program, clicking and editing various cells, going back and forth between the JTree and the JTable, to see why sharing an editor isn’t a good idea.

14.13 chap14.interlude.DemoFrame.java

```

1  package chap14.interlude;
2
3  import java.awt.BorderLayout;
4  import java.awt.GridLayout;
5
6  import javax.swing.BorderFactory;
7  import javax.swing.JFrame;
8  import javax.swing.JPanel;
9  import javax.swing.JScrollPane;
10 import javax.swing.JTable;
11 import javax.swing.JTree;
12 import javax.swing.table.DefaultTableModel;
13 import javax.swing.tree.DefaultMutableTreeNode;
14 import javax.swing.tree.DefaultTreeModel;
15
16 public class DemoFrame extends JFrame {
17
18     public DemoFrame() {
19         super("Tree and Table Editor Demo");
20
21         DemoTreeOrTableCellHandler handler1 = new DemoTreeOrTableCellHandler();
22
23         JTable table = createTable();
24         table.setDefaultRenderer(Object.class, handler1);
25         table.setDefaultEditor(Object.class, handler1);
26
27         DemoTreeOrTableCellHandler handler2 = new DemoTreeOrTableCellHandler();
28
29         JTree tree = createTree();
30         tree.setCellRenderer(handler2);
31         tree.setCellEditor(handler2);
32
33         //uncomment to see what happens.
34         //tree.setCellRenderer(handler1);
35         //tree.setCellEditor(handler1);
36
37         JPanel contentPane = new JPanel();
38         setContentPane(contentPane);
39         contentPane.setLayout(new GridLayout(1, 2));
40         JPanel treePanel = new JPanel(new BorderLayout());
41         JPanel tablePanel = new JPanel(new BorderLayout());
42         tablePanel.add(BorderLayout.CENTER, new JScrollPane(table));
43         tablePanel.setBorder(BorderFactory.createTitledBorder("Table Demo"));
44         treePanel.add(BorderLayout.CENTER, new JScrollPane(tree));
45         treePanel.setBorder(BorderFactory.createTitledBorder("Tree Demo"));
46         contentPane.add(treePanel);
47         contentPane.add(tablePanel);

```

```

48
49     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
50     pack();
51     setVisible(true);
52
53     }
54     private JTable createTable() {
55         Object[][] data = { { "Mickey", "NMI", "Mouse", "mousem" }, {
56             "Donald", "NMI", "Duck", "duckd" }, {
57             "Felix", "the", "Cat", "catf" }, {
58             "Winnie", "ther", "Pooh", "poohw" }
59         };
60         Object[] columns = { "First", "Middle", "Last", "User Name" };
61
62         DefaultTableModel model = new DefaultTableModel(data, columns);
63         JTable table = new JTable(model);
64
65         return table;
66     }
67     private JTree createTree() {
68         DefaultMutableTreeNode food = new DefaultMutableTreeNode("Food");
69         DefaultMutableTreeNode fruits = new DefaultMutableTreeNode("Fruits");
70         DefaultMutableTreeNode vegetables =
71             new DefaultMutableTreeNode("Vegetables");
72         DefaultMutableTreeNode apples = new DefaultMutableTreeNode("Apples");
73         DefaultMutableTreeNode pears = new DefaultMutableTreeNode("Pears");
74         DefaultMutableTreeNode cucumbers = new DefaultMutableTreeNode("Cucumbers");
75         DefaultMutableTreeNode tomatoes = new DefaultMutableTreeNode("Tomatoes");
76         food.add(fruits);
77         food.add(vegetables);
78         fruits.add(apples);
79         fruits.add(pears);
80         vegetables.add(cucumbers);
81         vegetables.add(tomatoes);
82
83         DefaultTreeModel model = new DefaultTreeModel(food);
84         JTree tree = new JTree(model);
85         tree.setEditable(true);
86
87         return tree;
88     }
89     public static void main(String[] arg) {
90         new DemoFrame().setVisible(true);
91     }
92 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap14/interlude/DemoFrame.java
java -cp classes chap14/interlude/DemoFrame

```

Quick Review

JTables, JTrees and JComboBoxes are able to edit their underlying data models, and they provide methods for plugging in custom editors through a mechanism that is similar to plugging in custom renderers. To create a custom editor for a JTree or JTable, it is easiest to extend `AbstractCellEditor` and implement either the `TreeCellEditor` or the `TableCellEditor` interface as required.

STEP 6 – A GOOD COMPONENT GETS BETTER

Swing's `JList` is a very useful component for displaying lists of data. However, as noted in the previous section, it doesn't provide a means for editing or manipulating its data model in any way. In this final step, we will write the new class, `DragList`, which extends `JList` to give it the ability to reorder its items by dragging one item at a time to a new position. We'll deal with a bit of offscreen graphics and transparency along with a good dose of old-fashioned cleverness. Reapplying a pattern set by an earlier section, we will create the `ReorderEvent` and `ReorderListener` classes. We will also create a mechanism to notify objects that implement `ReorderListener` and are registered with the `DragList` whenever a `ReorderEvent` occurs.

THE DRAGLIST CLASS – OVERVIEW

The `DragList` class (*example 14.14*) is more complex than other GUI classes we have written, so we will discuss its logic thoroughly.

ALGORITHM

When the user presses the mouse in a list item and begins to drag it, `DragList` sets the object, `dragItem`, to this item and creates an image of its corresponding list cell, `dragImage`. It erases that list cell (*by painting a blank rectangle over it*) suggesting that the dragged item has vacated its initial position. While the mouse is dragged, `DragList` tracks the mouse's motion, drawing `dragImage` semi-transparently over the contents of the list at the position of the mouse. Each time `dragImage` encroaches on a neighboring list cell, `DragList` switches `dragItem` and that list cell's corresponding value in the `ListModel`. This change of order is automatically and visibly manifested in the `DragList`. For as long as the mouse is dragged, the reordering process continues as necessary and `DragList` continues to erase the dragged list cell wherever it may be at the moment. When the mouse is finally released, `DragList` erases `dragImage` and stops erasing `dragItem`'s list cell, thereby creating the illusion that the dragged cell has finally come to rest. Registered `ReorderListeners` are then notified that the list has been reordered.

USE `DEFAULTLISTMODEL` (LINES 41 - 65)

Notice that the algorithm involves moving items' positions within the list. While `JList` doesn't provide methods for altering its data, it does provide access to its data model through the `getModel()` method which returns a `ListModel`. Unfortunately, the API for `ListModel` doesn't provide methods for altering its data either, so we need to find a `ListModel` implementation that is editable. We could, of course, write our own class that implements `ListModel` and also supports editing, but we don't need to. Starting with the javadocs for `ListModel` and searching via the "All Known Implementing Classes:" and "Direct Known Subclasses:" links we find that Swing provides the `DefaultListModel` class which provides a whole host of methods that support editing. The inheritance hierarchy for `DefaultListModel` is shown in Figure 14-10.

```
javax.swing.ListModel
    javax.swing.AbstractListModel
        javax.swing.DefaultListModel
```

Figure 14-10: `DefaultListModel` Inheritance Hierarchy

Because having an editable `ListModel` is a requirement for `DragList` to be functional, we should ensure that `DragList` never uses anything but a `DefaultListModel`. We provide two constructors: one that takes no arguments and just creates a `DefaultListModel` for itself, and another that accepts a `DefaultListModel` argument to be used as its model. We also need to override any methods available from its superclass, `JList`, that might set the model to something other than a `DefaultListModel`. This requires us to override three methods. We override `setModel(ListModel)` to throw a `ClassCastException` if the argument is not a `DefaultListModel`, and we override `setListData(Object[])` and `setListData(Vector)` to reuse `DragList`'s `DefaultListModel` by clearing it and adding the items in the array or `Vector` to it one by one.

REORDEREVENT AND REORDERLISTENER (LINES 184 - 203)

Just as we did earlier with `CheckboxListCell` by creating a new event and listener type, we now create the `ReorderEvent` and `ReorderListener` for `DragList`. The `ReorderEvent` class defines the event of reordering the items in the list. The `ReorderListener` interface defines the one method `listReordered(ReorderEvent)`. `ReorderEvent`'s only constructor takes a `DragList` as parameter so that it can set the event's source as that `DragList`. `DragList`'s `addReorderListener()` method adds a `ReorderListener` to a private list of listeners and its `removeReorderListener()` method removes the specified `ReorderListener` from the list of listeners. `DragList`'s `notifyListeners()` method constructs a `ReorderEvent` and sends it to all registered `ReorderListeners`.

THE DRAGLIST CLASS – DRAGGING

In the course of a mouse drag, three event types are always generated. These are in order:

1. a mouse press event when the user presses the mouse to begin the drag
2. a series of mouse drag events as the user drags the mouse
3. a mouse release event when the user releases the mouse thereby ending the drag

DragList implements `MouseListener` and writes `mousePressed()`, `mouseDragged()` and `mouseReleased()` event handlers that maintain several DragList attributes needed to achieve the visual effect of dragging. DragList registers itself as its own `MouseListener`.

dragIndex, dragItem and dragRect Attributes

The `mousePressed()` method initializes three attributes needed to create an image of the dragged item. These are `dragIndex`, `dragItem` and `dragRect` as shown in table 14-12.

Name	Meaning	How Obtained
<code>dragIndex</code>	The dragged item's index in the list.	Obtained by passing the point of the mouse press into <code>JList</code> 's <code>locationToIndex()</code> method.
<code>dragItem</code>	The item being dragged.	Obtained by passing <code>dragIndex</code> into the <code>ListModel</code> 's <code>getElementAt()</code> method.
<code>dragRect</code>	The rectangular area that the <code>DragList</code> will use to paint the dragged item.	Obtained by passing <code>dragIndex</code> into <code>JList</code> 's <code>getCellBounds()</code> method.

Table 14-12: `dragIndex`, `dragItem` and `dragRect` Attributes

dragStart, dragThreshold and allowDrag Attributes

The `mousePressed()` method initializes three attributes needed for the `mouseDragged()` method to determine whether an item is being dragged or not. These are `dragStart`, `dragThreshold` and `allowDrag` as shown in table 14-13.

Name	Meaning	Initialization
<code>dragStart</code>	The location of the mouse press.	Set to <code>MouseEvent.getPoint()</code> .
<code>dragThreshold</code>	The maximum vertical distance the mouse can be dragged before <code>dragItem</code> should be considered to be dragged. This eliminates unwanted drags due to inadvertent mouse jiggles.	Set to 1/4 the height of <code>dragRect</code> .
<code>allowDrag</code>	Whether or not dragging should be allowed.	Set to true if the mouse pressed event meets certain criteria.

Table 14-13: `dragStart`, `dragThreshold` and `allowDrag` Attributes

deltaY and inDrag Attributes

Two other attributes are needed to enable the dragging process. These are `deltaY` and `inDrag` as shown in table 14-14.

		Interested Methods		
Name	Meaning	mousePressed()	mouseDragged()	mouseReleased()
deltaY	Where, with respect to the mouse position, dragImage should be painted as the mouse is dragged.	Sets to the vertical distance between dragStart, and the top of the list item containing dragStart.	Uses deltaY to update dragRect's position ensuring that the top of dragRect is always the same vertical distance from the mouse location.	N/A
inDrag	Whether or not an item is in the process of being dragged.	N/A	Sets to true only if allowDrag is true, the drag is with the left-mouse button and the mouse has moved at least dragThreshold pixels vertically away from dragStart.	Sets to false.

Table 14-14: deltaY and inDrag Attributes

How the inDrag Attribute is Used

As shown in table 14-15, the paintComponent(), mouseDragged() and mouseReleased() methods do different things depending on the value of inDrag.

		Interested Methods		
Value	paintComponent()	mouseDragged()	mouseReleased()	
true	Calls super.paintComponent(), erases dragItem from the list and paints dragImage at the current mouse location.	Updates dragRect's position, updates the ListModel and calls repaint() when needed.	Sets inDrag to false, calls repaint() and notifies ReorderListeners.	
false	Calls super.paintComponent().	Calls createDragImage if it determines that inDrag should become true.	Does nothing.	

Table 14-15: How the inDrag Attribute is Used

THE DRAGLIST CLASS – PAINTING

Two methods handle the actual painting of the DragList. These are: createDragImage() which is called by mouseDragged() when mouseDragged() has just set inDrag to true; and paintComponent() which is called by the Swing painting mechanism when mouseDragged() and mouseReleased() call repaint().

THE CREATEDRAGIMAGE() METHOD (LINES 66 - 94)

When called, createDragImage() creates an image of dragItem and paints it into dragImage. It does this by obtaining the component that would be used to “rubber stamp” dragItem's image onto the list and using a handy SwingUtilities method to paint this component into dragImage. If dragImage is null or not the right size, createDragImage() first initializes dragImage to a new image of the correct size.

THE PAINTCOMPONENT() METHOD (LINES 95 - 108)

The `paintComponent()` method is called by the Swing painting mechanism in response to calls to `repaint()`. If `inDrag` is true, `paintComponent()` erases the area where `dragItem` was painted by filling `dragItem`'s rectangular area with the `DragList`'s background color. Then it sets the graphics context to 50% transparency and paints `dragImage` at the location of `dragRect`.

THE DRAGLIST CLASS – CODE

14.14 chap14.gui5.DragList.java

```

1      package chap14.gui5;
2
3      import java.awt.AlphaComposite;
4      import java.awt.Component;
5      import java.awt.Composite;
6      import java.awt.Graphics;
7      import java.awt.Graphics2D;
8      import java.awt.Insets;
9      import java.awt.Point;
10     import java.awt.Rectangle;
11     import java.awt.event.MouseEvent;
12     import java.awt.image.BufferedImage;
13     import java.util.EventListener;
14     import java.util.EventObject;
15     import java.util.Iterator;
16     import java.util.Vector;
17
18     import javax.swing.DefaultListModel;
19     import javax.swing.JList;
20     import javax.swing.ListCellRenderer;
21     import javax.swing.ListModel;
22     import javax.swing.SwingUtilities;
23     import javax.swing.event.MouseInputListener;
24
25     public class DragList extends JList implements MouseInputListener {
26
27         private Object dragItem;
28         private int dragIndex = -1;
29         private BufferedImage dragImage;
30         private Rectangle dragRect = new Rectangle();
31         private boolean inDrag = false;
32
33         private Point dragStart;
34         private int deltaY;
35         private int dragThreshold;
36
37         private boolean allowDrag;
38
39         private Vector listeners = new Vector();
40
41         public DragList() {
42             this(new DefaultListModel());
43         }
44         public DragList(DefaultListModel lm) {
45             super(lm);
46             addMouseListener(this);
47             addMouseMotionListener(this);
48         }
49         public void setModel(ListModel lm) {
50             super.setModel((DefaultListModel)lm);
51         }
52         public void setListData(Object[] listData) {
53             DefaultListModel model = (DefaultListModel)getModel();
54             model.clear();
55             for (int i = 0; i < listData.length; ++i) {
56                 model.addElement(listData[i]);
57             }
58         }
59         public void setListData(Vector listData) {
60             DefaultListModel model = (DefaultListModel)getModel();
61             model.clear();
62             for (Iterator it = listData.iterator(); it.hasNext(); ) {
63                 model.addElement(it.next());
64             }
65         }
66         private void createDragImage() {

```



```

67     if (dragImage == null
68         || dragImage.getWidth() != dragRect.width
69         || dragImage.getHeight() != dragRect.height) {
70         dragImage =
71             new BufferedImage(
72                 dragRect.width,
73                 dragRect.height,
74                 BufferedImage.TYPE_INT_RGB);
75     }
76     Graphics g = dragImage.getGraphics();
77
78     ListCellRenderer renderer = getCellRenderer();
79     Component comp =
80         renderer.getListCellRendererComponent(
81             this,
82             dragItem,
83             dragIndex,
84             true,
85             true);
86     SwingUtilities.paintComponent(
87         g,
88         comp,
89         this,
90         0,
91         0,
92         dragRect.width,
93         dragRect.height);
94 }
95 protected void paintComponent(Graphics g) {
96     super.paintComponent(g);
97     if (inDrag) {
98         Graphics2D g2 = (Graphics2D)g;
99         g2.setColor(getBackground());
100        Rectangle r = getCellBounds(dragIndex, dragIndex);
101        g2.fillRect(r.x, r.y, r.width, r.height);
102        Composite saveComposite = g2.getComposite();
103        g2.setComposite(
104            AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f));
105        g2.drawImage(dragImage, dragRect.x, dragRect.y, this);
106        g2.setComposite(saveComposite);
107    }
108 }
109 public void mousePressed(MouseEvent e) {
110     allowDrag = false;
111     if (!SwingUtilities.isLeftMouseButton(e)) {
112         return;
113     }
114     dragStart = e.getPoint();
115     dragIndex = locationToIndex(dragStart);
116     if (dragIndex < 0) {
117         return;
118     }
119     dragRect = getCellBounds(dragIndex, dragIndex);
120     if (!dragRect.contains(dragStart)) {
121         clearSelection();
122         return;
123     }
124     allowDrag = true;
125     DefaultListModel model = (DefaultListModel)getModel();
126     dragItem = model.getElementAt(dragIndex);
127     dragThreshold = dragRect.height / 4;
128     deltaY = dragStart.y - dragRect.y;
129 }
130 public void mouseDragged(MouseEvent e) {
131     if (!allowDrag) {
132         return;
133     }
134     Point mouse = e.getPoint();
135     if (!inDrag && Math.abs(mouse.y - dragStart.y) < dragThreshold) {
136         return;
137     }
138     if (!inDrag) {
139         clearSelection();
140         createDragImage();
141         inDrag = true;
142     }
143     //remember dragRect.y
144     int oldTop = dragRect.y;
145
146     dragRect.y = mouse.y - deltaY;
147     //dragRect is now at the accurate vertical location

```

```

148     //shift dragRect up or down as necessary so that it doesn't
149     //spill over the top or bottom of the DragList
150     Insets insets = getInsets();
151     dragRect.y = Math.max(dragRect.y, insets.top);
152     dragRect.y =
153         Math.min(dragRect.y, getHeight() - dragRect.height - insets.bottom);
154
155     //index is the index of the item located at the vertical center of dragRect
156     int index =
157         locationToIndex(new Point(mouse.x, dragRect.y + dragRect.height / 2));
158     if (dragIndex != index) {
159         DefaultListModel model = (DefaultListModel)getModel();
160         //move dragItem to the new location in the list
161         model.remove(dragIndex);
162         model.add(index, dragItem);
163         dragIndex = index;
164     }
165
166     int minY = Math.min(dragRect.y, oldTop);
167     int maxY = Math.max(dragRect.y, oldTop);
168     repaint(dragRect.x, minY, dragRect.width, maxY + dragRect.height);
169 }
170 public void mouseReleased(MouseEvent e) {
171     if (inDrag) {
172         setSelectedIndex(dragIndex);
173         inDrag = false;
174         repaint(dragRect);
175         notifyListeners(new ReorderEvent(this));
176     }
177 }
178
179 public void mouseClicked(MouseEvent e) {}
180 public void mouseMoved(MouseEvent e) {}
181 public void mouseEntered(MouseEvent e) {}
182 public void mouseExited(MouseEvent e) {}
183
184 public static class ReorderEvent extends EventObject {
185     public ReorderEvent(DragList dragList) {
186         super(dragList);
187     }
188 }
189 public static interface ReorderListener extends EventListener {
190     public void listReordered(ReorderEvent e);
191 }
192 public void addReorderListener(ReorderListener rl) {
193     listeners.add(rl);
194 }
195 public void removeReorderListener(ReorderListener rl) {
196     listeners.remove(rl);
197 }
198 private void notifyListeners(ReorderEvent event) {
199     for (Iterator it = listeners.iterator(); it.hasNext();) {
200         ReorderListener rl = (ReorderListener)it.next();
201         rl.listReordered(event);
202     }
203 }
204 }

```

THE MAINFRAME CLASS

Changes to MainFrame are straightforward. MainFrame implements DragList.ReorderListener, codes listReordered(DragList.ReorderEvent) to call redrawBoy() and declares the garmentList attribute to be a DragList instead of a JList. Just because this is the final step, its constructor also creates a slightly fancier interface.

14.15 chap14.gui5.MainFrame.java

```

1     package chap14.gui5;
2
3     import java.awt.BorderLayout;
4     import java.awt.Container;
5     import java.awt.Image;
6     import java.awt.Point;
7     import java.util.Collections;
8     import java.util.Vector;
9
10    import javax.swing.BorderFactory;
11    import javax.swing.JFrame;
12    import javax.swing.JLabel;
13    import javax.swing.JList;

```

```

14     import javax.swing.JPanel;
15     import javax.swing.ListModel;
16     import javax.swing.border.BevelBorder;
17
18     import utils.ResourceUtils;
19     import chap14.gui0.DressingBoard;
20     import chap14.gui0.Garment;
21     import chap14.gui4.CheckboxListCell;
22
23     public class MainFrame
24     extends JFrame
25     implements DragList.ReorderListener, CheckboxListCell.ToggleListener {
26
27     private DressingBoard dressingBoard;
28     private DragList garmentList;
29
30     public MainFrame() {
31         setTitle(getClass().getName());
32         setSize(600, 400);
33         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
34
35         JPanel topPanel = new JPanel(new BorderLayout());
36         topPanel.setBorder(BorderFactory.createEtchedBorder());
37
38         Container contentPane = getContentPane();
39         contentPane.setLayout(new BorderLayout());
40         contentPane.add("Center", topPanel);
41
42         JPanel mainPanel = new JPanel(new BorderLayout());
43         topPanel.add("Center", mainPanel);
44         String instructions =
45             "Click the checkboxes to put on or remove garments."
46             + " Drag the labels to determine order.";
47         topPanel.add("North", new JLabel(instructions, JLabel.CENTER));
48
49         dressingBoard = new DressingBoard();
50         mainPanel.add("Center", dressingBoard);
51
52         garmentList = new DragList();
53         garmentList.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
54         CheckboxListCell ccr = new CheckboxListCell() {
55             protected boolean getCheckedValue(Object value) {
56                 return ((Garment)value).isWorn();
57             }
58         };
59         ccr.addToggleListener(this);
60         garmentList.addMouseListener(ccr);
61         garmentList.setCellRenderer(ccr);
62         garmentList.addReorderListener(this);
63         mainPanel.add("West", garmentList);
64
65         initList();
66     }
67     private void initList() {
68         String[] names =
69             {
70                 "T-Shirt",
71                 "Briefs",
72                 "Left Sock",
73                 "Right Sock",
74                 "Shirt",
75                 "Pants",
76                 "Belt",
77                 "Tie",
78                 "Left Shoe",
79                 "Right Shoe" };
80         Point[] points =
81             {
82                 new Point(75, 125),
83                 new Point(86, 197),
84                 new Point(127, 256),
85                 new Point(45, 260),
86                 new Point(69, 118),
87                 new Point(82, 199),
88                 new Point(88, 203),
89                 new Point(84, 124),
90                 new Point(129, 258),
91                 new Point(40, 268)};
92
93         Vector garments = new Vector(names.length);
94         for (int i = 0; i < names.length; ++i) {

```

```

95     Image image =
96         ResourceUtils.loadImage("chap14/images/" + names[i] + ".gif", this);
97     Garment garment = new Garment(image, points[i].x, points[i].y, names[i]);
98     garments.add(garment);
99 }
100 Collections.shuffle(garments);
101 garmentList.setListData(garments);
102 }
103 public void listReordered(DragList.ReorderEvent e) {
104     redrawBoy();
105 }
106 public void checkboxToggled(CheckboxListCell.ToggleEvent event) {
107     JList list = (JList)event.getSource();
108     int index = event.getIndex();
109     Garment garment = (Garment)list.getModel().getElementAt(index);
110     garment.setWorn(!garment.isWorn());
111     redrawBoy();
112 }
113 private void redrawBoy() {
114     ListModel lm = garmentList.getModel();
115     int stop = lm.getSize();
116     Vector order = new Vector();
117     for (int i = 0; i < stop; ++i) {
118         Garment garment = (Garment)lm.getElementAt(i);
119         if (garment.isWorn()) {
120             order.add(garment);
121         }
122     }
123     dressingBoard.setOrder((Garment[])order.toArray(new Garment[0]));
124 }
125 public static void main(String[] arg) {
126     new MainFrame().setVisible(true);
127 }
128 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap14/gui5/MainFrame.java
java -cp classes chap14.gui5.MainFrame

```

We're done! Run the program. Have fun. In the process of creating this application you have gained experience with several advanced features of the Swing API. This is just the tip of the iceberg, but I hope it gives you a good feeling for the power Swing offers that is just waiting to be harnessed by a creative and determined mind.

SUMMARY

There are several ways to load an Image from a URL, the easiest being to use the `javax.imageio` package introduced with Java 1.4. When time comes to deploy an application, it is wise to package its classes and other resources into a jar file and to load the resources from a dynamically constructed URL.

The `Graphics` and `Graphics2D` classes provide many sophisticated drawing tools. Their methods can be divided into three categories, drawing operations, property-related methods and a few miscellaneous methods. The AWT/Swing framework handles the repainting of components. We can call one of a component's `repaint()` methods to trigger the framework to repaint our component in response to application specific needs. Application specific painting code should be placed into the `paint()` method of an AWT component and into the `paintComponent()` method of a Swing component.

Swing's separable model architecture creates a loose coupling between Components and the data they represent by defining data model interfaces with which the Components interact. This frees an application to manage its data as it wishes. As long as the data is wrapped in the appropriate interface, the Component will be automatically notified of changes in the data without any special effort on the part of the programmer.

`JList`, `JComboBox`, `JTree` and `JTable` are examples of faux-composite components that use renderers to create the illusion of having many children components without incurring the overhead of maintaining that many components. `JList` and `JComboBox` use `ListCellRenderers`, `JTables` use `TableCellRenderers` and `JTrees` use `TreeCellRenderers`. By implementing the correct renderer interface, a custom renderer can affect a faux-composite component's appearance, tailoring it to the data it represents.

JTables, JTrees and JComboBoxes are able to edit their underlying data models and they provide methods for plugging in custom editors through a mechanism that is similar to plugging in custom renderers. To create a custom editor for a JTree or JTable it is easiest to extend AbstractCellEditor and implement either the TableCellEditor or the TableCellEditor interface as required.

Skill-Building Exercises

- Skill with Coordinates:** Write a “joke” application that displays a JFrame containing a “Quit Application” button whose ActionListener quits the application. Unfortunately for the unsuspecting user, this button always moves away from the cursor just far enough to elude a mouseclick. No matter where the button moves, it must not move outside the bounds of the window. To be nice, make sure that pressing the “Enter” key will trigger the button’s actionPerformed() handler.
- Handling Images:** Extend JPanel to create a BackgroundPanel class. This class should allow an image to be used as the panel’s background. The panel should resize or tile its image (*depending on the mode*) as necessary to always completely fill its bounds. The image’s colors should be muted by a configurable amount through the use of transparency.
- Handling Images (Trickier):** Extend JList to create a BackgroundList class. This class should allow an image to be used as the JList’s background. The JList should resize or tile its image (*depending on the mode*) as necessary to always completely fill its bounds. The image colors should be muted by a configurable amount through the use of transparency. Whatever component serves as the ListCellRenderer component should have a transparent background so that the list’s own background shows through.
- Fine-Tuning the API:** The editor for the JTree in DemoFrame leaves something to be desired. To view the problem, run the DemoFrame application, edit a node in the tree by deleting all its text and press the “Enter” key to stop editing. Now try to edit it again by typing text back into it. Figure out what’s happening and implement a solution for it.
- Assimilating your Knowledge:** Modify the DemoTreeAndTable class to display a pop-up menu that appears when the user right-clicks a table row. This menu should provide a “Delete Row” menu item that deletes the clicked row, an “Insert Row Before” menu item that inserts a row before the clicked row and an “Insert Row After” menu item that inserts a row after the clicked row.
- Writing a DragTable:** Following the pattern set by DragList as much as possible, write a DragTable class that extends JTable and allows reordering of rows.

SUGGESTED PROJECTS

- Dragging with java.awt.dnd:** Investigate the java.awt.dnd package and see if you can write a DndList that allows reordering.
- Assorted Advanced Topics:** Investigate some of the articles at [java.sun.com/products/jfc/tsc/articles]
- Painting Program:** Write a small painting program. It should draw lines, circles and rectangles as determined by the user in response to mousePresses, mouseDrags and mouseReleases.

SELF-TEST QUESTIONS

1. What is double-buffering?
2. Where should you place custom painting code in an AWT component?
3. Where should you place custom painting code in a Swing component?
4. Explain what a Swing component's `paint()` method does.
5. Compare and contrast the use of `Component.paint()` with the use of the `Component.repaint()` methods.
6. What is Separable Model Architecture and what are its benefits?
7. What is a *faux-composite* component?
8. What is a *renderer* and which Swing faux-composite components use them?
9. What is an *editor* and which Swing faux-composite components use them?
10. What are the three categories of Graphics methods?
11. What is the difference between the `Graphics` and the `Graphics2D` classes?
12. How can one obtain a reference to a `Graphics2D` object?

REFERENCES

Amy Fowler, A Swing Architecture Overview: [java.sun.com/products/jfc/tsc/articles/architecture]

Java 2 API Specification: [java.sun.com/j2se/1.4.2/docs/api]

The Java Tutorial: [java.suc.com/docs/books/tutorial]

NOTES

PART IV: INTERMEDIATE CONCEPTS

CHAPTER 15



Reflections

EXCEPTIONS

LEARNING OBJECTIVES

- *DRAW AND DESCRIBE THE THROWABLE CLASS INHERITANCE HIERARCHY*
- *EXPLAIN THE PURPOSE AND USE OF ERRORS*
- *EXPLAIN THE PURPOSE AND USE OF EXCEPTIONS*
- *EXPLAIN THE PURPOSE AND USE OF RUNTIMEEXCEPTIONS*
- *EXPLAIN THE DIFFERENCE BETWEEN CHECKED AND UNCHECKED EXCEPTIONS*
- *STATE THE PURPOSE OF THE TRY, CATCH, AND FINALLY BLOCKS*
- *DEMONSTRATE THE USE OF A TRY/CATCH STATEMENT*
- *DEMONSTRATE THE USE OF A TRY/CATCH/FINALLY STATEMENT*
- *DEMONSTRATE THE USE OF A TRY/FINALLY STATEMENT*
- *DEMONSTRATE THE USE OF MULTIPLE CATCH BLOCKS*
- *DEMONSTRATE THE USE OF THE THROWS CLAUSE*
- *DEMONSTRATE HOW TO PERFORM EXCEPTION CHAINING*
- *DEMONSTRATE HOW TO EXTEND THE EXCEPTION CLASS TO CREATE CUSTOM EXCEPTIONS*
- *DEMONSTRATE YOUR ABILITY TO USE EXCEPTIONS EFFECTIVELY IN YOUR JAVA PROGRAMS*

INTRODUCTION

Exceptions are used in Java to indicate that an exceptional condition has occurred during program execution. You have been informally exposed to exceptions since chapter 3. The code examples presented thus far have minimized the need to explicitly handle exceptions. However, Java programs are always under the threat of throwing some kind of exception. (*By exception I mean any class that extends the `java.lang.Throwable` class. Two direct subclasses of `Throwable` are the `Error` and `Exception` classes, but I'm getting a little ahead of myself.*)

Some types of exceptions need to be handled explicitly by the programmer. These require the use of try/catch blocks. Other types of exceptions, specifically exceptions that extend the `Error` class and exceptions that extend the `RuntimeException` class, can be thrown by the Java Virtual Machine during program execution. Any attempt to catch and handle all these exception conditions would quickly exhaust you.

This chapter dives deep into the explanation of exceptions to show you how to properly handle existing exceptions (*i.e., those already defined by the Java API*) and how to create your own custom exceptions. You will be formally introduced to the `Throwable` class, try/catch/finally blocks, and the `throws` clause. You will also learn the difference between checked and unchecked exceptions. After reading this chapter you will be well prepared to properly handle exception conditions in your programs. You will also know how to create your own custom exception classes to better fit your abstraction requirements.

Why is this chapter placed at this point in this book? The answer is simple: you will soon reach the chapters that cover topics like network and file I/O programming. In these two areas especially you will be required to explicitly handle many types of exception conditions. As you write increasingly complex programs you will come to rely on the information conveyed by a thrown exception to help you debug your code. Now is the time to dive deep into the topic of exceptions!

THE THROWABLE CLASS HIERARCHY

The `Throwable` class is the daddy of all exceptions. Figure 15-1 illustrates the `Throwable` class hierarchy.

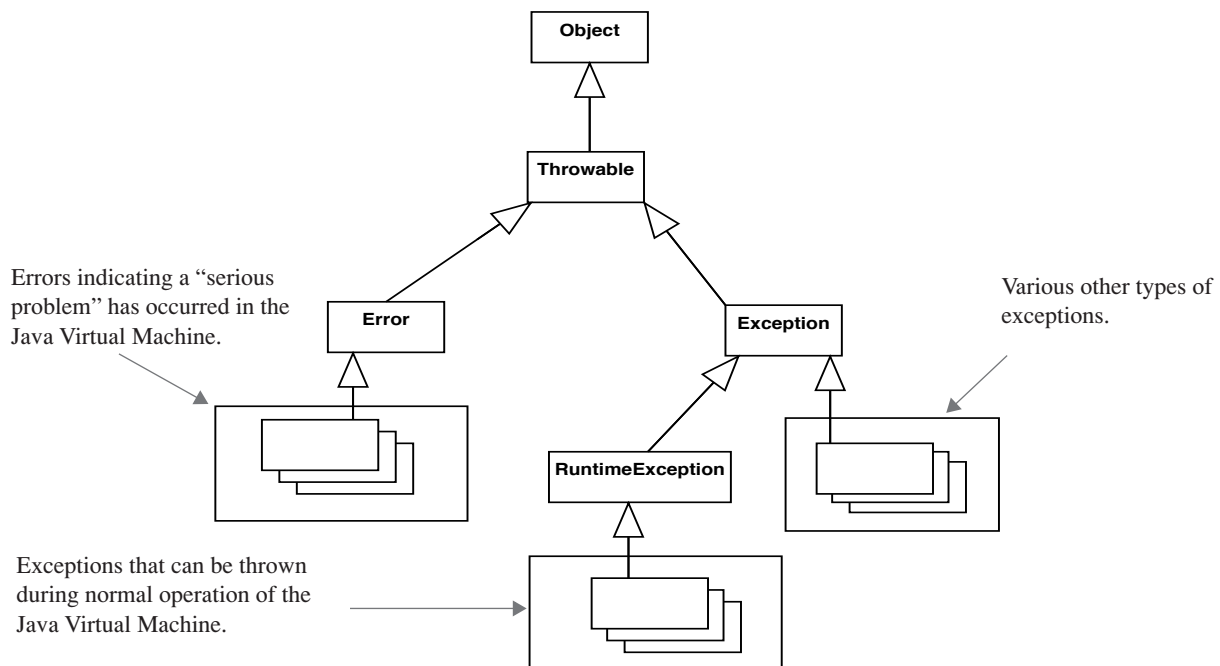


Figure 15-1: Throwable Class Hierarchy

Referring to figure 15-1 — the Throwable class extends Object as expected. Throwable has two direct subclasses: Error and Exception.

There are several types of Error subclasses and many types of Exception subclasses. Errors and Exceptions are each treated differently by the Java Virtual Machine and by programmers as you will soon learn. The RuntimeException class, and its subclasses, is also given special treatment of which you must be aware. Errors, Exceptions, and RuntimeExceptions are discussed in greater detail below.

ERRORS

Errors are thrown by the Java Virtual Machine in the event of a non-recoverable failure. The Java API documentation specifies that “An Error ... indicates serious problems that a reasonable application should not try to catch.”

Essentially, error conditions can occur at any time for reasons beyond the normal control of a programmer. What this means to you is that any method, of any class, at any time during its execution, might possibly throw an Error! Don't bother trying to catch them.

EXCEPTIONS

Exceptions are the opposite of Errors in that a reasonable application — meaning all the applications you write — will want to plan for them to occur and handle them properly in the source code. In fact, the Java compiler will ensure you address the possibility of an Exception being thrown by signaling a compiler error in cases where you have not done so. (*See Checked vs. Unchecked Exceptions below.*)

Generally speaking, you must explicitly address the possibility of a thrown Exception in your code unless the Exception is a RuntimeException.

RUNTIMEEXCEPTIONS

A RuntimeException represents a condition that may occur during the normal execution of the Java Virtual Machine. It is a direct subclass of Exception and it and its subclasses are given special treatment. Specifically, you do not have to explicitly handle the possibility of a thrown RuntimeException in your code. Take for example the NumberFormatException whose class inheritance hierarchy is shown in figure 15-2.

```
java.lang.Object
  |-java.lang.Throwable
    |-java.lang.Exception
      |-java.lang.RuntimeException
        |-java.lang.IllegalArgumentException
          |-java.lang.NumberFormatException
```

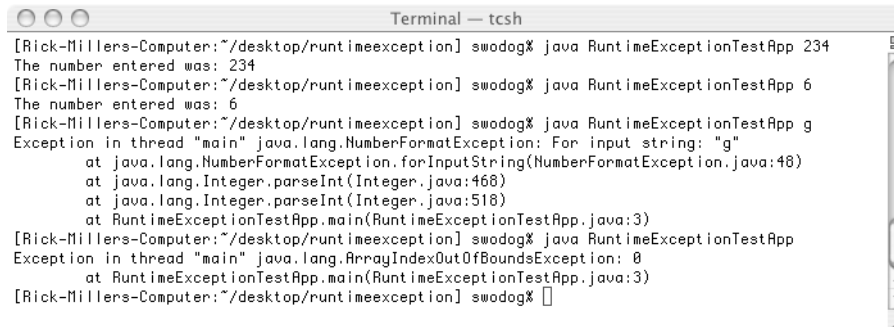
Figure 15-2: NumberFormatException Class Inheritance Hierarchy

Referring to figure 15-2 — NumberFormatException is a subclass of IllegalArgumentException, which in turn is a subclass of RuntimeException. The NumberFormatException is declared to be thrown by the Integer.parseInt() method. You have seen the Integer class, and its parseInt() method, used in code examples of previous chapters. Example 15.1 gives a short example to refresh your memory.

15.1 RuntimeExceptionTestApp.java

```
1 public class RuntimeExceptionTestApp {
2     public static void main(String[] args){
3         int i = Integer.parseInt(args[0]);
4         System.out.println("The number entered was: " + i);
5     }
6 }
```

In this short example the Integer.parseInt() method is used to parse a String entered via the command line and convert it into an integer. If the conversion is successful, the integer value is assigned to the variable i and its value is printed to the console. Notice the absence of a try/catch block — it's not required since NumberFormatException is a RuntimeException. Figure 15-2 shows the results of running example 15.1 with good and bad input strings.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/runtimeexception] swodog% java RuntimeExceptionTestApp 234
The number entered was: 234
[Rick-Millers-Computer:~/desktop/runtimeexception] swodog% java RuntimeExceptionTestApp 6
The number entered was: 6
[Rick-Millers-Computer:~/desktop/runtimeexception] swodog% java RuntimeExceptionTestApp g
Exception in thread "main" java.lang.NumberFormatException: For input string: "g"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:468)
    at java.lang.Integer.parseInt(Integer.java:518)
    at RuntimeExceptionTestApp.main(RuntimeExceptionTestApp.java:3)
[Rick-Millers-Computer:~/desktop/runtimeexception] swodog% java RuntimeExceptionTestApp
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at RuntimeExceptionTestApp.main(RuntimeExceptionTestApp.java:3)
[Rick-Millers-Computer:~/desktop/runtimeexception] swodog% █

```

Figure 15-3: Results of Running Example 15.1 with Good and Bad Input Strings

Referring to figure 15-3 — the application is run first with the character string “234” and again with the string “6”. Both times the application executes fine, converting the input string to an integer and printing the value. On the third run the character ‘g’ is entered on the command line and the program throws a `NumberFormatException` as is shown by the resulting stack trace printout. The program was run a fourth time with no string input. This resulted in another type of `RuntimeException` being thrown — `ArrayIndexOutOfBoundsException`.

As you can see from this short example `RuntimeExceptions` are given special treatment by the Java Virtual Machine.

But, just because you don’t need to catch `RuntimeExceptions` doesn’t mean you should never catch them or ignore them willy-nilly. The prudent programmer must make every effort to recover gracefully from program anomalies whether they are the result of bad user input or some other cause.

CHECKED vs. UNCHECKED EXCEPTIONS

Exceptions that extend from the `Exception` class, and are not `RuntimeExceptions`, are expected to be dealt with explicitly in the code by the programmer. These are referred to as checked exceptions because the Java compiler will ensure the `Exception` is being handled in some manner.

Errors and `RuntimeExceptions` do not need to be explicitly handled by the programmer and are therefore referred to as unchecked exceptions.

Quick Review

Errors and Exceptions are thrown by the Java Virtual Machine to indicate or signal the occurrence of an exceptional (abnormal) condition. The `Throwable` class is the root of all Errors and Exceptions.

Errors are thrown when a serious condition exists from which recovery is not possible. Errors are not handled by the programmer since they can potentially happen at any time under circumstances beyond their control.

Exceptions must be handled explicitly by the programmer unless the `Exception` is a `RuntimeException`. `RuntimeExceptions` can occur during the normal execution of the Java Virtual Machine. Although `RuntimeExceptions` can generally be ignored, the prudent programmer should give careful thought to their occurrence and handle them accordingly.

Classes that extend `Exception` are referred to as checked exceptions. The Java compiler will signal your failure to properly handle a checked exception. `RuntimeExceptions` and Errors are referred to as unchecked exceptions because programmers are not explicitly required to catch and handle them in their code.

Handling Exceptions

An exception object is “thrown” during the course of program execution to signal the occurrence of an exception condition. The object thrown will be of type `Throwable` and will either be an `Error` or `RuntimeException` (*i.e. unchecked exception*) or any other subtype of `Exception`. (*i.e. checked exception*) If you are writing code that might possibly throw a checked exception you must explicitly handle the thrown exception in some way. This section shows

you how to use the try/catch/finally statement and its several authorized combinations to properly handle exceptions in your programs.

TRY/CATCH STATEMENT

The try/catch statement, also referred to as a try/catch block, is used to properly handle an exception. The code that might throw the exception is placed in the try block and, if the exception is thrown, it is “caught” and handled in the catch block.

Let’s revisit example 15.1 and wrap the offending code in a try/catch block. Example 15.2 shows the modified code. The name of the class has been changed to TryCatchTestApp.

15.2 TryCatchTestApp.java

```

1      public class TryCatchTestApp {
2          public static void main(String[] args){
3              try{
4                  int i = Integer.parseInt(args[0]);
5                  System.out.println("The number entered was: " + i);
6              }catch(NumberFormatException e){
7                  System.out.println("The string you entered was not an integer!");
8              }
9          }
10     }

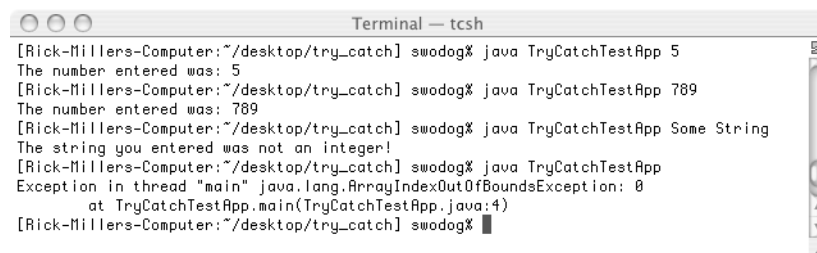
```

There are several items to note in this version of the program. First, line 4 contains the code that may throw a `NumberFormatException` if the `String` object referred to at `args[0]` fails to parse to an integer value. Although `NumberFormatException` is a `RuntimeException`, and is therefore an unchecked exception, it’s still a good idea to catch exceptions of this type in your code.

Line 5 now contains the code that prints the value of the variable `i` to the console. Line 5 will only execute if line 4 successfully executes. In other words, if the `NumberFormatException` is thrown during the execution of line 4, the remaining lines of code in the try block will be skipped and program execution will jump to the code contained in the catch block. (*If the exception thrown matches the exception type expected in the catch block.*)

The catch block begins on line 6. The type of exception, in this case `NumberFormatException`, is used to declare a parameter for the catch clause named `e`. In this example the parameter `e` is not used inside the catch block, however, a diagnostic message is printed to the console informing the user that the string they entered was not an integer.

Figure 15-4 shows the results of running example 15.2.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/try_catch] swadog% java TryCatchTestApp 5
The number entered was: 5
[Rick-Millers-Computer:~/desktop/try_catch] swadog% java TryCatchTestApp 789
The number entered was: 789
[Rick-Millers-Computer:~/desktop/try_catch] swadog% java TryCatchTestApp Some String
The string you entered was not an integer!
[Rick-Millers-Computer:~/desktop/try_catch] swadog% java TryCatchTestApp
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at TryCatchTestApp.main(TryCatchTestApp.java:4)
[Rick-Millers-Computer:~/desktop/try_catch] swadog% █

```

Figure 15-4: Results of Running Example 15.2

Referring to figure 15-4 — The program appears to behave better now that the `NumberFormatException` is explicitly handled in a try/catch block. The program is run first with the string “5”, next with the string “789”, and a third time with the string “Some String” with expected results. However, if a user fails to enter a text string when running the program it will still throw the `ArrayIndexOutOfBoundsException`. We can fix this problem by adding another catch block to catch this type of exception.

Multiple Catch Blocks

It is often the case that your program runs the risk of throwing several different types of exceptions. More often than not you also want to provide some sort of custom error handling depending on the type of exception thrown. Multiple catch blocks can be used after a try block to handle this situation.

Example 15.3 gives an improved version of the previous example. The name of the class has been changed to `MultipleCatchTestApp`.

15.3 *MultipleCatchTestApp.java*

```

1      public class MultipleCatchTestApp {
2          public static void main(String[] args){
3              try{
4                  int i = Integer.parseInt(args[0]);
5                  System.out.println("The number entered was: " + i);
6              }catch(NumberFormatException e){
7                  System.out.println("The string you entered was not an integer!");
8              }
9              catch(ArrayIndexOutOfBoundsException e){
10                 System.out.println("You must enter a string! -- Please try again!");
11             }
12         }
13     }

```

Notice in this example that the second catch block was added just after the first catch block. The first catch block is skipped if its exception parameter type fails to match that of the thrown exception. In this case the thrown exception is then compared to the parameter type in the second catch block. If it's a match the second catch block code is executed. If the thrown exception fails to match any of the expected exception types then the exception is propagated up the Java Virtual Machine calling stack in the hope that it will eventually find a block of code that will properly handle the exception. In this simple example there's not much farther up the chain an exception can be deferred until the Java Virtual Machine handles it by printing its detail to the console.

THE IMPORTANCE OF EXCEPTION TYPE AND ORDER IN MULTIPLE CATCH BLOCKS

When using multiple catch blocks to catch different types of exceptions you must be aware of the types of exceptions you are trying to catch and the order in which they are caught.

The order in which you catch exceptions must go from the specific type to the general type. Since exceptions are objects that belong to a class hierarchy they behave like ordinary objects. This means that subclass objects can be substituted where base class objects are expected. For instance, if a catch clause parameter specifies an exception of type `Exception` then it will also catch and handle all subtypes of `Exception`. If you use multiple catch blocks and want to catch a subtype of `Exception` then the subtype (specific) catch block must appear before the base type (general) catch block.

Luckily, if you reverse the order and try to catch the general type of exception before the specific type the Java compiler will signal your error.

Example 15.4 shows how to catch a specific type of exception followed by a general type of exception.

15.4 *GoodCatchBlockOrderingTestApp.java*

```

1      public class GoodCatchBlockOrderingTestApp {
2          public static void main(String[] args){
3              try{
4                  int i = Integer.parseInt(args[0]);
5                  System.out.println("The number entered was: " + i);
6              }catch(NumberFormatException e){
7                  System.out.println("The string you entered was not an integer!");
8              }
9              catch(Exception e){
10                 System.out.println("You must enter a string! -- Please try again!");
11             }
12         }
13     }

```

This example looks like the previous example except the class name has been changed and the catch block on line 9 now catches `Exception` instead of `ArrayIndexOutOfBoundsException`. The behavior of this program is somewhat different because any `Exception` thrown from the try block, other than `NumberFormatException`, will be caught and handled in the second catch block.

You could achieve the same effect by substituting the class `Throwable` for the class `Exception` in the catch clause on line 9. In this way you would catch not only thrown `Exceptions` but `Errors` as well. (*i.e. All Throwable objects!*)

Generally speaking, you will want to catch and handle the most specific exception appropriate to your programming situation. If catching the general exception is good enough, so be it, however, finer levels of error handling are usually required to produce production-quality code.

MANIPULATING A THROWABLE OBJECT

So far in this chapter I have only showed you how to print a simple diagnostic message to the console after an exception has been caught. In this section I want to show you what you can do with a Throwable object inside the catch block.

You can do several things with a Throwable object after it's been caught. These actions are listed in the following bullets:

- You could just rethrow the object but this is of limited value.
- You can repackage (*chain*) the Throwable object and throw the new Throwable object. This is good to do if you want to hide low-level Exceptions from higher level programmatic abstractions.
- You can perform some processing then rethrow the object.
- You can display detail information about the Throwable object.
- You can perform some processing and throw a new type of Throwable object.
- You can perform a combination of these actions as required.
- Or you can just catch and ignore the exception. (*Not recommended for production code.*)

Table 15-1 lists and summarizes several Throwable methods you will find immediately helpful. For a complete listing of Throwable methods consult the Java API documentation referenced at the end of this chapter.

Method	Description
<code>Throwable()</code>	Default constructor.
<code>Throwable(String message)</code>	Constructor that lets you create a Throwable object with a message String.
<code>Throwable(String message, Throwable cause)</code>	Constructor that lets you create a Throwable object with both a message string and a Throwable cause object. The cause object allows Exception chaining in Java 1.4 and above.
<code>Throwable(Throwable cause)</code>	Constructor that lets you create a Throwable object with a Throwable cause object. The Throwable object's message field will be set using <code>cause.toString()</code> or to null if the cause object is null.
<code>void printStackTrace()</code>	Prints the Throwable object's stacktrace to the standard error stream. The standard error stream is the console by default.
<code>String toString()</code>	Returns a short description of the Throwable object.
<code>String getMessage()</code>	Returns the String object containing the message details of this Throwable object.
<code>Throwable initCause(Throwable cause)</code>	Initializes the cause field of the Throwable object.
<code>Throwable getCause()</code>	Returns the Throwable object referred to by the cause field.

Table 15-1: Helpful Throwable Methods

The first four methods listed in table 15-1 are constructors. This means that there are four ways to create a Throwable object. You will see some of these constructor methods utilized in the Throwing Exceptions section later in this chapter.

You should also be aware that specific exception types may provide additional functionality appropriate to the type of exception being thrown. Exceptions of this type are usually found in the `java.io` and `java.net` packages. The best way to learn about the special behavior of the exceptions you are dealing with is to consult the Java Platform API documentation.

Example 15.5 expands on the previous example and shows you how to use several of the Throwable methods to extract detailed information about a caught exception.

15.5 *ExceptionDetailTestApp.java*

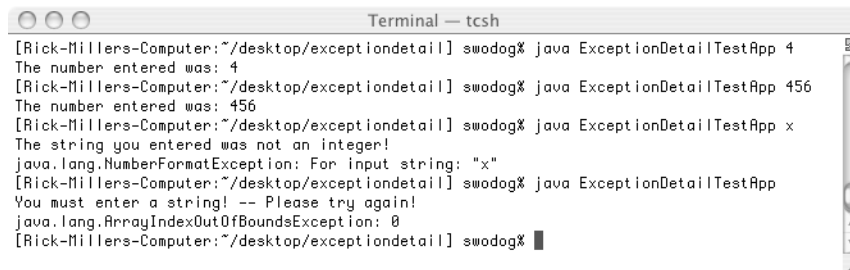
```

1      public class ExceptionDetailTestApp {
2          public static void main(String[] args){
3              try{
4                  int i = Integer.parseInt(args[0]);
5                  System.out.println("The number entered was: " + i);
6              }catch(NumberFormatException e){
7                  System.out.println("The string you entered was not an integer!");
8                  System.out.println(e.toString());
9                  // e.printStackTrace();
10             }
11             catch(Exception e){
12                 System.out.println("You must enter a string! -- Please try again!");
13                 System.out.println(e.toString());
14                 // e.printStackTrace();
15             }
16         }
17     }

```

This program is similar to example 15.4 with the exception of a class name change and the addition of four lines of code. On lines 8 and 9, located in the first catch block, the `NumberFormatException` parameter reference `e` is utilized to call the `toString()` and the `printStackTrace()` methods. (*Line 9 is presently commented out.*) The same is done on lines 13 and 14 of the second catch block. (*Line 14 is presently commented out as well.*)

Figure 15-5 shows the results of running this program.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/exceptiondetail] swodog% java ExceptionDetailTestApp 4
The number entered was: 4
[Rick-Millers-Computer:~/desktop/exceptiondetail] swodog% java ExceptionDetailTestApp 456
The number entered was: 456
[Rick-Millers-Computer:~/desktop/exceptiondetail] swodog% java ExceptionDetailTestApp x
The string you entered was not an integer!
java.lang.NumberFormatException: For input string: "x"
[Rick-Millers-Computer:~/desktop/exceptiondetail] swodog% java ExceptionDetailTestApp
You must enter a string! -- Please try again!
java.lang.ArrayIndexOutOfBoundsException: 0
[Rick-Millers-Computer:~/desktop/exceptiondetail] swodog% █

```

Figure 15-5: Results of Running Example 15.5

Referring to figure 15-5 — the program is run with four different inputs: “4”, “456”, “x”, and null or no string. The level of exception detail provided by the `toString()` method is brief, as can be seen in the program output.

Let’s run a different version of this program to see the difference between the `toString()` and the `printStackTrace()` methods. Example 15.6 gives the modified code.

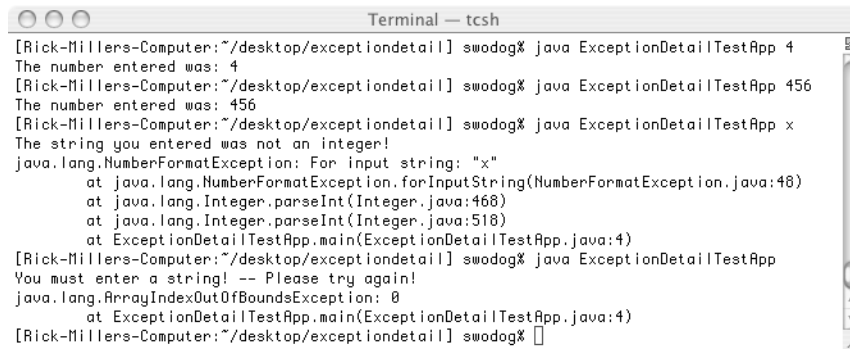
15.6 *ExceptionDetailTestApp.java (Mod 1)*

```

1      public class ExceptionDetailTestApp {
2          public static void main(String[] args){
3              try{
4                  int i = Integer.parseInt(args[0]);
5                  System.out.println("The number entered was: " + i);
6              }catch(NumberFormatException e){
7                  System.out.println("The string you entered was not an integer!");
8                  //System.out.println(e.toString());
9                  e.printStackTrace();
10             }
11             catch(Exception e){
12                 System.out.println("You must enter a string! -- Please try again!");
13                 //System.out.println(e.toString());
14                 e.printStackTrace();
15             }
16         }
17     }

```

The difference between this and the previous example is that lines 8 and 13 are now commented out. This will give us an example of the `printStackTrace()` method in action. Figure 15-6 shows the results of running this program. Compare figure 15-6 with figure 15-5 above and note the differences between calling the `toString()` and `printStackTrace()` methods. The `printStackTrace()` method gives more detail concerning the nature of the exception. If you don’t need this level of diagnostic output in your program use the `toString()` method.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/exceptiondetail] swodog% java ExceptionDetailTestApp 4
The number entered was: 4
[Rick-Millers-Computer:~/desktop/exceptiondetail] swodog% java ExceptionDetailTestApp 456
The number entered was: 456
[Rick-Millers-Computer:~/desktop/exceptiondetail] swodog% java ExceptionDetailTestApp x
The string you entered was not an integer!
java.lang.NumberFormatException: For input string: "x"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:468)
    at java.lang.Integer.parseInt(Integer.java:518)
    at ExceptionDetailTestApp.main(ExceptionDetailTestApp.java:4)
[Rick-Millers-Computer:~/desktop/exceptiondetail] swodog% java ExceptionDetailTestApp
You must enter a string! -- Please try again!
java.lang.ArrayIndexOutOfBoundsException: 0
    at ExceptionDetailTestApp.main(ExceptionDetailTestApp.java:4)
[Rick-Millers-Computer:~/desktop/exceptiondetail] swodog%

```

Figure 15-6: Results of Running Example 15.6

Try/Catch/Finally Statement

A finally clause can be added to the try/catch statement to provide a means to guarantee the execution of a block of code in the event an exception occurs. Finally blocks are used when valuable computer resources must be freed up and released back to the operating system. This usually occurs when you are doing file I/O or network programming.

Example 15.7 expands on the previous example and demonstrates the use of a finally block.

15.7 FinallyBlockTestApp.java

```

1      public class FinallyBlockTestApp {
2          public static void main(String[] args){
3              try{
4                  int i = Integer.parseInt(args[0]);
5                  System.out.println("The number entered was: " + i);
6                  if(i == 0){
7                      System.exit(0);
8                  }
9              }catch(NumberFormatException e){
10                 System.out.println("The string you entered was not an integer!");
11                 //System.out.println(e.toString());
12                 e.printStackTrace();
13             }
14             catch(Exception e){
15                 System.out.println("You must enter a string! -- Please try again!");
16                 //System.out.println(e.toString());
17                 e.printStackTrace();
18             }
19             finally{
20                 System.out.println("The code in the finally block is guaranteed to execute!");
21             }
22         }
23     }

```

Referring to example 15.7 — this program differs from the previous example in three ways: 1) the class name has been changed, 2) the finally block has been added starting on line 19, and 3) if the user enters a “0” on the command line when they run the program the if statement on line 6 will evaluate to true which will result in the System.exit() method being called on line 7. If one line of code in the try block is executed the code in the finally block is guaranteed to execute so long as the try block is not exited via a System.exit() method call. Figure 15-7 shows the results of running this program.

Referring to figure 15-7 — the program is run with several valid and invalid string inputs. Each time an exception is thrown the code in the finally block executes. However, when “0” is entered the execution of the Java Virtual Machine is stopped with a call to the System.exit() method.

Try/Finally Statement

As you saw earlier in this section the finally block can be omitted if one or more catch clauses follow a try block. Likewise, the catch clause can be omitted if a finally block is used in conjunction with a try block. However, with every try block you must use at least one catch clause or one finally clause as they both cannot be simultaneously omitted.

```

Terminal — tcsh

[Rick-Millers-Computer:~/desktop/finallyblock] swodog% java FinallyBlockTestApp 4
The number entered was: 4
The code in the finally block is guaranteed to execute!
[Rick-Millers-Computer:~/desktop/finallyblock] swodog% java FinallyBlockTestApp 235
The number entered was: 235
The code in the finally block is guaranteed to execute!
[Rick-Millers-Computer:~/desktop/finallyblock] swodog% java FinallyBlockTestApp k
The string you entered was not an integer!
java.lang.NumberFormatException: For input string: "k"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:468)
    at java.lang.Integer.parseInt(Integer.java:518)
    at FinallyBlockTestApp.main(FinallyBlockTestApp.java:4)
The code in the finally block is guaranteed to execute!
[Rick-Millers-Computer:~/desktop/finallyblock] swodog% java FinallyBlockTestApp
You must enter a string! -- Please try again!
java.lang.ArrayIndexOutOfBoundsException: 0
    at FinallyBlockTestApp.main(FinallyBlockTestApp.java:4)
The code in the finally block is guaranteed to execute!
[Rick-Millers-Computer:~/desktop/finallyblock] swodog% java FinallyBlockTestApp 0
The number entered was: 0
[Rick-Millers-Computer:~/desktop/finallyblock] swodog% █

```

Figure 15-7: Results of Running Example 15.7

When can you use the try/finally block combination? Good question. Generally speaking you can use the try/finally block combination anytime you want to guarantee a section of code will execute based upon the execution of some other section of code. Namely, if the code located in the try block executes then the code in the finally block is guaranteed to execute. So, in this regard, your imagination is the limit. However, in normal Java programming you don't usually see try/finally blocks utilized in this fashion.

A more common use of the try/finally block would be in a situation where you intended not to catch a thrown exception (*i.e. you will let it propagate up the calling stack*) but if the exception is thrown you want to do some housekeeping nonetheless.

Quick Review

Code that might throw a checked exception must be contained in a try block unless the method in which it appears uses the throws clause to indicate it might throw the exception. (*The use of the throws clause is covered in the next section.*) Failure to properly handle a checked exception will result in a compiler error. Code that might throw an unchecked exception can optionally be contained in a try block. A try block must be accompanied by either one or more catch blocks, a finally block, or a combination of both.

The Throwable class provides some useful methods for getting information about the nature of an exception. Some exceptions provide additional information about themselves. This is especially true of exceptions belonging to the java.io and java.net packages. Consult the Java Platform API documentation for detailed information about specific exception types.

THROWING EXCEPTIONS

In the previous section I showed you how to catch and process exceptions. In this section I will show you how to throw exceptions. You will want to throw an exception in several situations. For instance, if you don't want to handle an exception in a particular piece of code you must indicate to someone using the method in which your code appears that it will throw an exception. In another example, you may process the exception in your code, repackage or translate the exception into a different kind of exception, and throw the new exception.

THE THROWS CLAUSE

The throws clause is used in a method declaration or definition to indicate that the method in question might throw an exception of the indicated type(s). Examples 15.8 and 15.9 together demonstrate the use of the throws clause. This program takes a string input from the command line and attempts to load the class by that name, create

an object of the class type, and call a method on the newly created object. This program uses the services of the `java.lang.Class` class. Example 15.8 is a user-defined class named `ExampleClass` that provides simple functionality by overriding the `Object.toString()` method. Example 15.9 is an application named `ClassLoaderTestApp` that uses the `Class.forName()` method to dynamically load the `ExampleClass` and create an object of its type.

15.8 *ExampleClass.java*

```

1     public class ExampleClass {
2         public ExampleClass(){
3             System.out.println("ExampleClass object created!");
4         }
5         public String toString(){
6             return super.toString();
7         }
8     }

```

15.9 *ClassLoaderTestApp.java*

```

1     public class ClassLoaderTestApp {
2
3         Object _object = null;
4
5         public void loadClass(String class_name) throws ClassNotFoundException, InstantiationException,
6             IllegalAccessException, ArrayIndexOutOfBoundsException {
7             Class loaded_class = Class.forName(class_name);
8             _object = loaded_class.newInstance();
9             System.out.println(_object.toString());
10        }
11
12        public static void main(String[] args){
13            ClassLoaderTestApp clta = new ClassLoaderTestApp();
14            try{
15                clta.loadClass(args[0]);
16            }
17            catch(ClassNotFoundException e){
18                System.out.println("A class by that name does not exist. Please try again.");
19            }
20
21            catch(InstantiationException e){
22                System.out.println("Problem creating an object of that class type. Please try again.");
23            }
24            catch(IllegalAccessException e){
25                System.out.println(e.toString());
26            }
27            catch(ArrayIndexOutOfBoundsException e){
28                System.out.println("You failed to enter a string when you ran the program!"
29                    + "Please try again!");
30            }
31        }
32    }

```

Referring to example 15.9 — `ClassLoaderTestApp` has one method named `loadClass()` which uses the `throws` clause to indicate that its execution might result in four exceptions being thrown. All the exceptions are checked exceptions except our old friend `ArrayIndexOutOfBoundsException`.

The `loadClass()` method is called in the `main()` method, and it's the main method's responsibility to handle the exceptions thrown by the `loadClass()` method. (*Alternatively, the main() method could itself use the throws clause to pass the buck on up the calling chain.*)

When this program is run an instance of `ClassLoaderTestApp` is created and accessed via the reference named `clta`. The `loadClass()` method is called on line 15 using the `String` reference located at `args[0]` as an argument. In the `loadClass()` method, an attempt is made on line 7 to load the class whose name matches that of the input string. If this line fails, a `ClassNotFoundException` is thrown. If the class loads fine line 8 executes, however, if there's a problem creating an object of that class type an `InstantiationException` is thrown. If the object is created successfully then its string representation is printed to the console on line 9.

Figure 15-8 shows the results of running this program with valid and invalid input strings.

THE THROW KEYWORD

The `throw` keyword is used to explicitly throw an exception. The use of the `throw` keyword is discussed and demonstrated in detail in the `Creating Custom Exceptions` section.

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/ThrowsClause] swodog% java ClassLoaderTestApp ExampleClass
ExampleClass object created!
ExampleClass@54172f
[Rick-Millers-Computer:~/desktop/ThrowsClause] swodog% java ClassLoaderTestApp BadClassName
A class by that name does not exist. Please try again.
[Rick-Millers-Computer:~/desktop/ThrowsClause] swodog% java ClassLoaderTestApp
You failed to enter a string when you ran the program! Please try again!
[Rick-Millers-Computer:~/desktop/ThrowsClause] swodog% java ClassLoaderTestApp java.lang.Object
java.lang.Object@571886
[Rick-Millers-Computer:~/desktop/ThrowsClause] swodog% java ClassLoaderTestApp javax.swing.JButton
javax.swing.JButton[,0,0,0x0,invalid,layout=javax.swing.OverlayLayout,alignmentX=0.0,alignmentY=0.5,border=apple.laf.AquaButtonBorder@53ba3d,flags=296,maximumSize=,minimumSize=,preferredSize=,defaultIcon=,disabledIcon=,disabledSelectedIcon=,margin=javax.swing.plaf.InsetsUIResource[top=3,left=14,bottom=3,right=14],paintBorder=true,paintFocus=true,pressedIcon=,rolloverEnabled=false,rolloverIcon=,rolloverSelectedIcon=,selectedIcon=,text=,defaultCapable=true]
[Rick-Millers-Computer:~/desktop/ThrowsClause] swodog%

```

Figure 15-8: Results of Running Example 15.9

Quick Review

Use the throws clause in a method declaration to indicate what type(s) of exception(s) a method might throw.

CREATING CUSTOM EXCEPTIONS

You can easily create custom exceptions by extending any of the Throwable classes. For most situations you will extend the Exception class. In this case you would be creating a checked exception. You may discover a need to extend the RuntimeException class in which case you would be creating an unchecked exception, but a need to do this does not normally present itself. You would most certainly never find a need to extend the Error class unless perhaps you were writing your own custom Java Virtual Machine.

In this section I will show you how to extend the functionality of the Exception class to create your own custom exceptions. Here you'll learn how to use the throw keyword to explicitly throw an exception, how to translate low-level exceptions into higher level exception abstractions, how to chain exceptions, and how to set and get an exception's cause field.

High & Low Level Abstractions

The Java Platform API provides many types of exceptions that represent the range of exceptional conditions that can occur through the use of various API class methods. These exception abstractions are appropriate for the Java Platform API. However, if you are creating your own set of custom classes the exception abstractions provided by the Java API may be too low level to sufficiently represent the abstraction(s) you have in mind. When this situation arises you can create your own custom exceptions.

EXTENDING THE EXCEPTION CLASS

To create a custom exception you can extend the Exception class and add any functionality you require including additional fields and methods. Normally, however, you are well served by simply utilizing the functionality provided by the Exception class. Examples 15.10 and 15.11 demonstrate the use of a custom exception. Example 15.10 gives a class named CustomException that has four constructor methods, one for each type of constructor offered in the Exception class. Example 15.11 gives a modified version of the ClassLoaderTestApp program.

```

1      public class CustomException extends Exception {
2          public CustomException(String message, Throwable cause){
3              super(message, cause);
4          }
5
6          public CustomException(String message){
7              super(message);
8          }

```

15.10 CustomException.java

```

9
10     public CustomException(){
11         super("Custom Exception");
12     }
13
14     public CustomException(Throwable cause){
15         super(cause);
16     }
17 }

```

15.11 *ClassLoaderTestApp.java (Mod I)*

```

1     public class ClassLoaderTestApp {
2
3         Object _object = null;
4
5         public void loadClass(String class_name) throws CustomException {
6             try{
7                 Class loaded_class = Class.forName(class_name);
8                 _object = loaded_class.newInstance();
9                 System.out.println(_object.toString());
10            }
11            catch(ClassNotFoundException e){
12                throw new CustomException("A class by that name does not exist.", e);
13            }
14            catch(InstantiationException e){
15                throw new CustomException("Problem creating an object of that class type.", e);
16            }
17            catch(IllegalAccessException e){
18                throw new CustomException(e.toString(), e);
19            }
20        }
21
22
23        public static void main(String[] args){
24            ClassLoaderTestApp clta = new ClassLoaderTestApp();
25            try{
26                clta.loadClass(args[0]);
27            }
28            catch(CustomException e){
29                System.out.println(e.getMessage());
30                System.out.println("Caused by: " + e.getCause().toString());
31            }
32            catch(ArrayIndexOutOfBoundsException e){
33                System.out.println("You failed to enter a string! Please try again!");
34            }
35        }
36    }

```

Referring to example 15.11 — several important modifications were made to this program to demonstrate the use of the `CustomException`. First, the offending code in the `loadClass()` method is now enclosed in a try block. The try block is followed by three catch blocks to handle each type of exception that might be thrown when the code executes. Each catch block catches its particular exception and chains it to a new `CustomException` with a customized message. The newly created `CustomException` is thrown with the `throw` keyword.

The `main()` method code is now mostly concerned with catching `CustomExceptions`. However, no loss of information occurs because you can get the low level cause of the exception in addition to its message. Figure 15-9 shows the results of running this program.

```

Terminal - tcsh
[rick-millers-computer:book_code_examples/chapter_15/customexception] swadog% java ClassLoaderTestApp
You failed to enter a string! Please try again!
[rick-millers-computer:book_code_examples/chapter_15/customexception] swadog% java ClassLoaderTestApp CustomException
CustomException: Custom Exception
[rick-millers-computer:book_code_examples/chapter_15/customexception] swadog% java ClassLoaderTestApp java.lang.Object
java.lang.Object@723d7c
[rick-millers-computer:book_code_examples/chapter_15/customexception] swadog%

```

Figure 15-9: Results of Running Example 15.11

Referring to figure 15-9 — the `ClassLoaderTestApp` program is run first without an argument naming a class to dynamically load. This results in the expected error message being printed to the console. The second time around it is run with the string “`CustomException`”. This causes the `CustomException` class to be dynamically loaded and its

information printed to the console. The third time the program runs it's given the string "java.lang.Object" and, as expected, an Object is dynamically loaded and its information is printed to the console.

SOME ADVICE ON NAMING YOUR CUSTOM EXCEPTIONS

In this section I am completely guilty of asking you to do as I say but not as I just did! In the previous example I used the name CustomException for the sake of simplicity. In practice you will want to choose the names of your exception abstractions as carefully as you choose any of your abstraction names. This you must do to preserve the clarity of your program.

Quick Review

Create custom exceptions by extending the Exception class. Choose the names of your exception abstractions as carefully as you choose other program abstraction names.

You can chain low level exceptions to higher level exception abstractions by catching the low level abstraction in a catch block and using it as a cause argument for a custom exception. You can gain access to an exception's cause via the Throwable.getCause() method.

SUMMARY

Errors and Exceptions are thrown by the Java Virtual Machine to indicate or signal the occurrence of an exceptional (*abnormal*) condition. The Throwable class is the root of all Errors and Exceptions.

Errors are thrown when a serious condition exists from which recovery is not possible. Errors are not handled by the programmer since they can potentially happen at any time under circumstances beyond their control.

Exceptions must be handled explicitly by the programmer unless the Exception is a RuntimeException. RuntimeExceptions can occur during the normal execution of the Java Virtual Machine. Although RuntimeExceptions can generally be ignored, the prudent programmer should give careful thought to their occurrence and handle them accordingly.

Classes that extend Exception are referred to as checked exceptions. The Java compiler will signal your failure to properly handle a checked exception. RuntimeExceptions and Errors are referred to as unchecked exceptions because programmers are not explicitly required to catch and handle them in their code.

Code that might throw a checked exception must be contained in a try block unless the method in which it appears uses the throws clause to indicate it might throw the exception. Failure to properly handle a checked exception will result in a compiler error. Code that might throw an unchecked exception can optionally be contained in a try block. A try block must be accompanied by either one or more catch blocks, a finally block, or a combination of both.

The Throwable class provides some useful methods for getting information about the nature of an exception. Some exceptions provide additional information about themselves. This is especially true of exceptions belonging to the java.io and java.net packages. Consult the Java Platform API documentation for detailed information about specific exception types.

Use the throws clause in a method declaration to indicate what type(s) of exception(s) a method might throw.

Create custom exceptions by extending the Exception class. Choose the names of your exception abstractions as carefully as you choose other program abstraction names.

You can chain low-level exceptions to higher level exception abstractions by catching the low-level abstraction in a catch block and using it as a cause argument for a custom exception. You can gain access to an exception's cause via the Throwable.getCause() method.

Skill-Building Exercises

1. **API Research:** Access the online documentation for the Java 2 Standard Edition Platform API (version 1.4.2 or 5) and research all the exceptions contained therein. Start by visiting the Throwable class documentation and go from there. Pay particular attention to any special functionality a particular Exception offers over and above that provided by the Throwable class.
2. **UML Exercise:** Draw a UML diagram of the Throwable class inheritance hierarchy. You can use figure 15-1 as a guide. Label and describe the purpose of each of the primary classes.

SUGGESTED PROJECTS

None

SELF-TEST QUESTIONS

1. What is the purpose of an exception?
2. What class serves as the root of all Errors and Exceptions?
3. What's the difference between a checked and an unchecked exception?
4. (True/False) You must explicitly handle an unchecked exception.
5. What's the purpose of a try/catch statement?
6. How many catch clauses can a try/catch statement contain?
7. What's the purpose of the finally clause?
8. (True/False) Both the catch and finally clauses can be simultaneously omitted from a try/catch/finally statement.
9. What's the purpose of a throws clause?
10. In what order should you catch exceptions?

REFERENCES

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*, 4th Edition. O'Reilly and Associates, Inc. Sebastopol, CA. ISBN: 0-596-00283-1

James Gosling, et. al. *The Java™ Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, MA. ISBN: 0-201-31008-2

Jon Meyer, et. al. *Java™ Virtual Machine*. O'Reilly and Associates, Inc. Sebastopol, CA. ISBN: 1-56592-194-1

The Java 2 Standard Edition Version 5 Platform API Online Documentation [<http://java.sun.com/j2se/1.5.0/docs/api/index.html>]

The Java 2 Standard Edition Version 1.4.2 Platform API Online Documentation [<http://java.sun.com/j2se/1.4.2/docs/api/index.html>]

NOTES

CHAPTER 16



SNOW RUNNERS

THREADS

LEARNING OBJECTIVES

- *STATE THE DEFINITION OF A THREAD*
- *LIST AND DESCRIBE THE TWO WAYS TO CREATE AND START A THREAD*
- *LIST AND DESCRIBE THE DIFFERENCES BETWEEN CREATING A THREAD CLASS AND IMPLEMENTING THE RUNNABLE INTERFACE*
- *EXPLAIN HOW TO SET THE PRIORITY OF A THREAD*
- *EXPLAIN HOW THREAD PRIORITY BEHAVIOR DEPENDS ON THE OPERATING SYSTEM*
- *UNDERSTAND THE CAUSE OF RACE CONDITIONS*
- *UNDERSTAND THE MECHANICS OF SYNCHRONIZATION*
- *DEMONSTRATE ABILITY TO MANAGE MULTIPLE THREADS SHARING COMMON RESOURCES*
- *DEMONSTRATE ABILITY TO USE THE JAVA WAIT AND NOTIFY MECHANISM TO ENSURE COOPERATION BETWEEN THREADS*
- *UNDERSTAND THE CAUSE OF DEADLOCK CONDITIONS*

INTRODUCTION

You might never need to write a program that creates threads. You might accomplish many tasks in complete ignorance of the topic and of the fact that threads have been working in the background all along, enabling your programs to run at all. On the other hand, there may come a day when you write a program that performs some time-consuming operation and after it starts, you realize that you and your program have just been taken hostage by the operation — not able to do anything else (*including hit the “Cancel” button*) until the operation has come to its inevitable conclusion. Hopefully, that day (*if not earlier*) you will remember some of what you read in this chapter, and cast off those shackles of ignorance.

Writing threads is an exciting part of Java programming because it can free your approach to programming and allow you to think in terms of smaller, independent tasks working in harmony rather than one monolithic application. Along with an understanding of threads-programming, however, comes an awareness of the dark side of threads. The challenges associated with managing thread behavior can be quite thorny and will lead us into an exploration of some of the most mysterious corners of the Java language.

WHAT IS A THREAD?

A thread is a sequence of executable lines of code. Without your necessarily being aware of it, every program we have written has been a thread. When the JVM runs an application, it creates a thread called “main” that executes the application step by step. This thread begins its existence at the opening curly bracket of the public static `main(String[])` method. It executes the lines of code as they progress, branching, looping, invoking methods on objects, etc., however the program directs. While it has lines of code to execute, a thread is said to be “running” and it remains in existence. The “main” thread stops running and ends its existence when it runs past the edge of the last curly bracket of the main method.

If we think of our own human body as a “computer”, then an example of a thread in everyday life might be *walking*. The walking thread begins when you stand up intending to go somewhere. While you are walking, the thread is “running” (*pardon the pun*). The walking thread ends when you reach your destination and sit down. Another thread might be *eating*, which begins when you sit down to eat, is running while you continue to chew, swallow and possibly get seconds, and ends when you get up from the table. Or how about a conversation thread which begins when you start conversing with someone, is running as you talk and listen, and ends when both people have stopped talking and listening to each other?

Your body is multi-threaded in that it can and does run more than one thread in parallel or *concurrently*. Some concurrent threads can run quite independently, having minimal impact on each other. For example, you can converse with someone with whom you are walking because you walk with your legs and converse with your mouth and ears. Other threads are not so independent. For example, conversing and eating. They can and often do run concurrently, but because they both require your mouth’s full participation, there are times when you have to choose between swallowing or speaking. At those times, an attempt to do both simultaneously could have undesirable consequences.

The computer, like our body, is multi-threading. For instance, it can download a file from the internet while you are starting up your photo-editing program. It can play an audio CD while you are saving changes to a document. It can even start up two different applications at the same time. But, as with our bodies, there are some things a computer cannot do concurrently like saving changes to a document that is in the process of being deleted. Last but not least, the JVM is also multi-threading, which means that you can write Java programs that create and run multiple threads. A Java program may create any number of threads with their own independent execution paths, sharing system resources as well as objects created by the program itself.

Threads are instances of the class `java.lang.Thread`. They are organized into `ThreadGroups`. In addition to containing `Threads`, `ThreadGroups` may contain other `ThreadGroups`. This tree-like organization exists primarily as a means of supporting security policies that might restrict access from one `ThreadGroup` to another, but `ThreadGroups` also simplify the management of threads by providing methods that affect all the threads within a group. When an application is first started, the “main” `ThreadGroup` is created and given one `Thread` (*named “main”*) whose execution path is the “main” method. The application is then free to create and run any number of additional application-

specific Threads. When a Thread is constructed, it is contained by the ThreadGroup specified in its constructor, or, if none was specified, by the ThreadGroup that contains the currently running Thread.

We will be discussing threads in greater detail soon, but first let's consider the following program which does nothing at all except to print out all the active threads. Don't worry how the utility method works, but please pay attention to the output. Figure 16-1 shows the console output from running example 16.1.

```

1      package chap16.simple;
2
3      import utils.TreePrinterUtils;
4
5      public class Main {
6          public static void main(String[] args) {
7              TreePrinterUtils.printThreads();
8          }
9      }

```

16.1 chap16.simple.Main.java

Use the following commands to compile and execute the example. Ensure you have compiled the contents of the Book_Code_Examples/utils folder and placed the resulting class files in your classpath. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/simple/Main.java
java -cp classes chap16.simple.Main

```

```

ThreadGroup "system" (Maximum Priority: 10)
+--Thread "Reference Handler" (Priority: 10) - Daemon
+--Thread "Finalizer" (Priority: 8) - Daemon
+--Thread "Signal Dispatcher" (Priority: 10) - Daemon
+--Thread "CompileThread0" (Priority: 10) - Daemon
+--ThreadGroup "main" (Maximum Priority: 10)
    +--Thread "main" (Priority: 5)

```

Figure 16-1: Results of Running Example 16.1

This minimal program has five threads and two ThreadGroups! At the top of the tree structure is the “system” thread group. It contains four threads as well as the “main” thread group. The “main” thread group contains the “main” thread which was created by the application. The other threads were created by the system to support the JVM. Along with the names of the threads and thread groups, the utility method lists thread and thread group priorities inside parenthesis. Thread priority will be discussed later. If the thread is a daemon thread, that is stated as well. Unlike normal threads, the existence of a running daemon thread doesn't prevent the JVM from exiting.

Now let's consider the following minimal GUI program that prints out all the active threads when a button is clicked. While it is running, click the “Print Threads” button to print out all the active threads.

```

1      package chap16.simple;
2
3      import java.awt.BorderLayout;
4      import java.awt.Container;
5      import java.awt.event.ActionEvent;
6      import java.awt.event.ActionListener;
7
8      import javax.swing.JButton;
9      import javax.swing.JFrame;
10
11     import utils.TreePrinterUtils;
12
13     public class Gui extends JFrame {
14         public Gui() {
15             Container contentPane = getContentPane();
16             JButton button = new JButton("Print Threads");
17             button.addActionListener(new ActionListener() {
18                 public void actionPerformed(ActionEvent e) {
19                     TreePrinterUtils.printThreads();
20                 }
21             });

```

16.2 chap16.simple.Gui.java

```

22         contentPane.add(button, BorderLayout.CENTER);
23         pack();
24     }
25     public static void main(String[] args) {
26         new Gui().show();
27     }
28 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/simple/Gui.java
java -cp classes chap16.simple.Gui

```

A click of the “Print Threads” button produces the output shown in figure 16-2 when running in Microsoft Windows 2000 Professional. (*Note: The output may be slightly different if running Apple’s OS X.*)

```

ThreadGroup "system" (Maximum Priority: 10)
+--Thread "Reference Handler" (Priority: 10) - Daemon
+--Thread "Finalizer" (Priority: 8) - Daemon
+--Thread "Signal Dispatcher" (Priority: 10) - Daemon
+--Thread "CompileThread0" (Priority: 10) - Daemon
+--ThreadGroup "main" (Maximum Priority: 10)
    +--Thread "AWT-Shutdown" (Priority: 5)
    +--Thread "AWT-Windows" (Priority: 6) - Daemon
    +--Thread "Java2D Disposer" (Priority: 10) - Daemon
    +--Thread "AWT-EventQueue-0" (Priority: 6)
    +--Thread "DestroyJavaVM" (Priority: 5)

```

Figure 16-2: Results of Running Example 16.2

The “system” ThreadGroup hasn’t changed, but notice that the “main” ThreadGroup contains five threads – none of which is the “main” Thread! These five threads were created by the JVM to support the AWT/Swing framework. What happened to the “main” Thread? Well, as soon as it executed its last line (*line 26*), it was finished executing its lines of code so it died a natural death. Unless a Swing application explicitly creates new threads, once its main method has completed, the rest of its life will be hosted by the nine threads you see above.

Quick Review

A thread is a sequence of executable lines of code. Threads are organized into a tree structure composed of threads and thread groups. Java programs can create and run more than one thread concurrently. All Java programs are multi-threaded because the JVM and the Swing/AWT framework are themselves multi-threaded.

Building a Clock Component

Before we delve too deeply into the mechanics of threads, let’s start by thinking how we might create a Component that displays the current time every second. Our goal will be to write a self-contained Component that we might place into the interface of an application to display the current time to the nearest second. It should run in the background meaning that it shouldn’t take up all the computer’s CPU time. Getting the time and displaying it in a Component are simple enough, but how are we going to do it *every second*?

Example 16.3 prints the time to System.out every second but it’s not ideal. Before you run it, be aware that this program is an infinite loop, so you will have to manually kill the process that started it. Your development environment should provide a simple way to do this. Alternatively, Control-C (*in Unix-based machines*) or closing the command prompt window that started the process (*in Windows*) should do the trick.

```

1     package chap16.clock;
2
3     import java.text.DateFormat;
4     import java.util.Date;
5
6     public class Clock1 {
7         public static void main(String[] arg) {
8             Date date = new Date();
9             DateFormat dateFormatter = DateFormat.getTimeInstance();
10            String lastDateString = "";
11
12            while (true) {
13                long currentMillis = System.currentTimeMillis();
14                date.setTime(currentMillis);
15                String curDateString = dateFormatter.format(date);
16                if (!lastDateString.equals(curDateString)) {
17                    lastDateString = curDateString;
18                    System.out.println(curDateString);
19                }
20            }
21        }
22    }

```

Use the following commands to compile the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/clock/Clock1.java
java -cp classes chap16.clock.Clock1

```

Will this program’s logic work for our background clock? No, it won’t because it keeps the CPU way too busy. No sooner does it compute the current time, compare it to the last computed time and perhaps print it to screen, than it immediately repeats the process. My computer goes through this compute/compare process more than 200,000 times before the second has changed and the current time needs to be printed again. This is quite a lot of effort for very little result. Unless it can rest for approximately one second between iterations, this thread will be less of a background process than a foreground process CPU hog. We will find a better way in the next section.

CURRENTTHREAD(), SLEEP(), INTERRUPT(), INTERRUPTED() AND ISINTERRUPTED()

The Thread class provides the static method, `currentThread` (listed in table 16-1), which returns a reference to the currently executing thread. The currently executing thread is necessarily the thread that is calling “`Thread.currentThread()`”. (Think about that if it didn’t make sense right away!) Many Thread methods are static because they are designed to affect only the current thread. In other words, invoking “`whateverThread.someStaticMethod()`” is equivalent to invoking “`Thread.currentThread().someStaticMethod()`” which is equivalent to invoking “`Thread.someStaticMethod()`”. The latter is the preferred way to call a static method as it minimizes possible ambiguity.

Method Name and Purpose
public static Thread currentThread() Returns a reference to the currently executing thread.

Table 16-1: Getting the Current Thread

`Thread.sleep()` is one of those static methods that operate only on the current thread. Pass it a length of time measured in milliseconds, with an optional parameter of nanoseconds, and the current thread will cease execution until the time has elapsed or until it has been interrupted by another thread. Methods for sleeping and interrupting threads are listed in table 16-2.

Method Name and Purpose
public static void sleep(long millis) throws InterruptedException Causes the current thread to sleep for the specified number of milliseconds or until it has been interrupted.

Table 16-2: Sleeping and Interrupting

<p>public static void sleep(long millis, int nanos) throws InterruptedException Causes the current thread to sleep for the specified number of milliseconds and additional nanoseconds or until it has been interrupted.</p>
<p>public void interrupt() Interrupts a thread that is sleeping or in another “waiting” mode. Otherwise it sets the thread’s interrupted status.</p>

Table 16-2: Sleeping and Interrupting

A thread interrupts another thread by invoking `interrupt()` on that thread as in:

```
threadToInterrupt.interrupt();
```

If the interrupted thread was sleeping when it was interrupted, the `sleep()` method will throw an `InterruptedException` which the thread can catch and deal with as desired. If it wasn’t sleeping when it was interrupted, then the interrupted thread’s interrupted status will be set. A thread can check its interrupted status at any time by calling its `isInterrupted()` method or, if it’s the current thread, by calling the static `interrupted()` method. Calling `interrupted()` clears the current thread’s interrupted status. Calling `isInterrupted()` does not clear a thread’s interrupted status. These two methods are listed in table 16-3.

Method Name and Purpose
<p>public boolean isInterrupted() Returns whether or not this thread has been interrupted. Does not clear its interrupted status.</p>
<p>public static boolean interrupted() Returns whether or not the current thread has been interrupted and clears its interrupted status if it was set.</p>

Table 16-3: Checking the Interrupted Status

The following clock program (*example 16.4*) uses `Thread.sleep()` to tell the current thread to wait for the number of milliseconds until the time will have reached the next second before repeating the process. This is the logic we will need for a background clock.

16.4 chap16.clock.Clock2.java

```

1      package chap16.clock;
2
3      import java.text.DateFormat;
4      import java.util.Date;
5
6      public class Clock2 {
7          public static void main(String[] arg) {
8              Date date = new Date();
9              DateFormat dateFormatter = DateFormat.getTimeInstance();
10
11             while (true) {
12                 long currentMillis = System.currentTimeMillis();
13                 date.setTime(currentMillis);
14                 String curDateString = dateFormatter.format(date);
15                 System.out.println(curDateString);
16                 long sleepMillis = 1000 - (currentMillis % 1000);
17                 try {
18                     Thread.sleep(sleepMillis);
19                 } catch (InterruptedException e) {}
20             }
21         }
22     }

```

Use the following commands to compile the example. From the directory containing the `src` folder:

```
javac -d classes -sourcepath src src/chap16/clock/Clock2.java
java -cp classes chap16.clock.Clock2
```

CREATING YOUR OWN THREADS

The thread class implements the Runnable interface which defines one method called run() that takes no parameters, returns no values and, as the javadocs state, “may take any action whatsoever”. There are basically two ways to create a Thread. One is to extend Thread and override its run() method to do whatever the thread is supposed to do. The other is to implement Runnable in another class, defining its run() method to do whatever the thread is supposed to do, and pass an instance of that class into the Thread’s constructor. If it was constructed with a Runnable parameter, Thread’s run() method will call the Runnable’s run() method. Optional parameters in the Thread constructors include the ThreadGroup to contain the Thread and a name for the Thread. All Threads have names which need not be unique. If a Thread is not assigned a name through its constructor, a name will be automatically generated for it. Except for the “main” thread which is started automatically by the JVM, all application created threads must be started by calling Thread.start(). This schedules the thread for execution. The JVM executes a thread by calling its run() method. Thread’s constructors are listed in table 16-4 and its start() and run() methods are listed in table 16-5.

Constructor Method Names
public Thread()
public Thread(Runnable target)
public Thread(ThreadGroup group, Runnable target)
public Thread(String name)
public Thread(ThreadGroup group, String name)
public Thread(Runnable target, String name)
public Thread(ThreadGroup group, Runnable target, String name);

Table 16-4: Thread Constructors

Method Name and Purpose
public void start() Causes the JVM to begin execution of this thread’s run method.
public void run() There are no restrictions on what action the run method may take.

Table 16-5: Starting a Thread

In the next program (*examples 16.5 and 16.6*), we extend Thread to define the ClockThread class. ClockThread’s constructor requires a Component parameter so that its run() method can call repaint() on it every second.

ClockPanel1 is a JPanel containing a JLabel. Its paintComponent() method is overridden so that whenever it is called, it will set the JLabel’s text to the current time. Its constructor creates an instance of ClockThread, passing itself as the component parameter, and starts the ClockThread. ClockPanel1’s “main” method constructs a simple JFrame that sports a running clock (*a ClockPanel1*) at the top and a JTextArea in its center. Each time the ClockThread calls repaint() on the ClockPanel1, the Swing framework is triggered to call paint(), which calls paintComponent() causing the displayed time to be updated.

16.5 chap16.clock.ClockThread.java

```

1      package chap16.clock;
2
3      import java.awt.Component;
4
5      public class ClockThread extends Thread {
6          private Component component;
7
8          public ClockThread(Component component) {
9              this.component = component;
10             setName("Clock Thread");

```



```

11     }
12     public void run() {
13         while (true) {
14             long currentMillis = System.currentTimeMillis();
15             long sleepMillis = 1000 - (currentMillis % 1000);
16             component.repaint();
17             try {
18                 Thread.sleep(sleepMillis);
19             } catch (InterruptedException e) {}
20         }
21     }
22 }

```

16.6 chap16.clock.ClockPanel1.java

```

1     package chap16.clock;
2
3     import java.awt.BorderLayout;
4     import java.awt.Graphics;
5     import java.text.DateFormat;
6     import java.util.Date;
7
8     import javax.swing.JFrame;
9     import javax.swing.JLabel;
10    import javax.swing.JPanel;
11    import javax.swing.JScrollPane;
12    import javax.swing.JTextArea;
13
14    public class ClockPanel1 extends JPanel {
15        private final JLabel clockLabel;
16        private final DateFormat dateFormatter = DateFormat.getTimeInstance();
17        private final Date date = new Date();
18        private final Thread clockThread;
19
20        public ClockPanel1() {
21            clockLabel = new JLabel("Clock warming up...", JLabel.CENTER);
22            setLayout(new BorderLayout());
23            add(clockLabel, BorderLayout.CENTER);
24            clockThread = new ClockThread(this);
25            clockThread.start();
26        }
27        public void paintComponent(Graphics g) {
28            updateTime();
29            super.paintComponent(g);
30        }
31        private void updateTime() {
32            long currentMillis = System.currentTimeMillis();
33            date.setTime(currentMillis);
34            String curDateString = dateFormatter.format(date);
35            clockLabel.setText(curDateString);
36        }
37        public static void main(String[] arg) {
38            JFrame f = new JFrame();
39            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
40            f.getContentPane().add(new ClockPanel1(), BorderLayout.NORTH);
41            f.getContentPane().add(
42                new JScrollPane(new JTextArea(20, 50)),
43                BorderLayout.CENTER);
44
45            f.pack();
46            f.show();
47        }
48    }

```

Use the following commands to compile and run the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/clock/ClockPanel1.java
java -cp classes chap16.clock.ClockPanel1

```

ClockPanel2 (*example 16.7*) accomplishes the same thing as ClockPanel1, but without extending Thread. It implements Runnable instead. ClockPanel2's constructor creates a new Thread object, passing itself in as the Runnable, and then starts the thread. Its run() method is identical to ClockThread's run() method except that it calls repaint() on itself rather than another component.

16.7 chap16.clock.ClockPanel2.java

```

1     package chap16.clock;
2
3     import java.awt.BorderLayout;
4     import java.awt.Graphics;

```

```

5     import java.text.DateFormat;
6     import java.util.Date;
7
8     import javax.swing.JFrame;
9     import javax.swing.JLabel;
10    import javax.swing.JPanel;
11    import javax.swing.JScrollPane;
12    import javax.swing.JTextArea;
13
14    public class ClockPanel2 extends JPanel implements Runnable {
15        private final JLabel clockLabel;
16        private final DateFormat dateFormatter = DateFormat.getTimeInstance();
17        private final Date date = new Date();
18        private final Thread clockThread;
19
20        public ClockPanel2() {
21            clockLabel = new JLabel("Clock warming up...", JLabel.CENTER);
22            setLayout(new BorderLayout());
23            add(clockLabel, BorderLayout.CENTER);
24            clockThread = new Thread(this);
25            clockThread.setName("Clock Thread");
26            clockThread.start();
27        }
28        public void paintComponent(Graphics g) {
29            updateTime();
30            super.paintComponent(g);
31        }
32        private void updateTime() {
33            long currentMillis = System.currentTimeMillis();
34            date.setTime(currentMillis);
35            String curDateString = dateFormatter.format(date);
36            clockLabel.setText(curDateString);
37        }
38        public void run() {
39            while (true) {
40                long currentMillis = System.currentTimeMillis();
41                long sleepMillis = 1000 - (currentMillis % 1000);
42                repaint();
43                try {
44                    Thread.sleep(sleepMillis);
45                } catch (InterruptedException e) {}
46            }
47        }
48        public static void main(String[] arg) {
49            JFrame f = new JFrame();
50            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51            f.getContentPane().add(new ClockPanel2(), BorderLayout.NORTH);
52            f.getContentPane().add(
53                new JScrollPane(new JTextArea(20, 50)),
54                BorderLayout.CENTER);
55
56            f.pack();
57            f.show();
58        }
59    }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/clock/ClockPanel2.java
java -cp classes chap16.clock.ClockPanel2

```

Take time to run `ClockPanel1` and/or `ClockPanel2`. Notice how the clock's behavior is concurrent with and independent of user activity in the `JTextArea`. Incidentally, once the "main" thread has finished its work, the only threads remaining are the system threads, the Swing threads and the one thread running the clock.

Quick Review

A thread can be created by extending `Thread` and overriding its `run` method or by passing a `Runnable` into one of the standard `Thread` constructors. A thread must be started by calling its `start()` method. Calling `Thread.sleep()` causes a thread to stop executing for a certain length of time and frees the CPU for other threads. A sleeping thread can be woken by calling `interrupt()`.

Computing Pi

We will create and run two threads in the next program. One of them will operate a Pi generator with a really cool algorithm that can generate Pi to an unlimited number of digits one at a time. The other thread will be the thread that runs inside a `ClockPanel1`. `PiPanel1` (example 16.8) creates a window with a `ClockPanel1` at the top, a `JTextArea` in the center that displays Pi continuously appending the latest-generated digits, and a `JButton` at the bottom that can pause or play the Pi-generation. There is some significant code in `PiPanel1` dealing with displaying Pi and scrolling the `JTextArea` as necessary so the latest digits are always in view. This code has nothing to do with threads and, for this chapter's purposes, you may ignore it. On the other hand it might give you some insight into the handling of `JTextAreas`, `JScrollPanes` and `JScrollBars`, so it's your choice. The `PiSpigot` class that generates Pi (example 16.9) is also not relevant to threads but it's based on such an incredible algorithm that we will discuss it at the end of this chapter for readers that are interested.

16.8 *chap16.pi.PiPanel1.java*

```

1      package chap16.pi;
2
3      import java.awt.BorderLayout;
4      import java.awt.Font;
5      import java.awt.event.ActionEvent;
6      import java.awt.event.ActionListener;
7
8      import javax.swing.JButton;
9      import javax.swing.JFrame;
10     import javax.swing.JPanel;
11     import javax.swing.JScrollBar;
12     import javax.swing.JScrollPane;
13     import javax.swing.JTextArea;
14     import javax.swing.SwingConstants;
15     import javax.swing.SwingUtilities;
16
17     public class PiPanel1 extends JPanel implements ActionListener {
18         private JTextArea textArea;
19         private JButton spigotButton;
20         private PiSpigot spigot;
21         private int numDigits = 0;
22         private JScrollBar vScrollBar;
23         private JScrollPane scrollPane;
24
25         private boolean paused = true;
26
27         /* producerThread is protected so that a subclass can reference it. */
28         protected Thread producerThread;
29
30         /* buttonPanel is protected so that a subclass can reference it. */
31         protected final JPanel buttonPanel;
32
33         public PiPanel1() {
34             setLayout(new BorderLayout());
35
36             textArea = new JTextArea(20, 123);
37             textArea.setFont(new Font("Monospaced", Font.PLAIN, 12));
38             textArea.setEditable(false);
39             scrollPane = new JScrollPane(textArea);
40             scrollPane.setHorizontalScrollBarPolicy(
41                 JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
42             scrollPane.setVerticalScrollBarPolicy(
43                 JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
44             add(scrollPane, BorderLayout.CENTER);
45
46             vScrollBar = scrollPane.getVerticalScrollBar();
47
48             spigotButton = new JButton("Play");
49             spigotButton.addActionListener(this);
50             buttonPanel = new JPanel(new BorderLayout());
51             buttonPanel.add(spigotButton, BorderLayout.CENTER);
52             add(buttonPanel, BorderLayout.SOUTH);
53
54             spigot = new PiSpigot();
55             startProducer();
56         }
57
58         /* Thread-related methods */
59         private void startProducer() {

```

```

60     producerThread = new Thread() {
61         public void run() {
62             while (true) {
63                 Thread.yield();
64                 while (paused) {
65                     pauseImpl();
66                 }
67                 int digit = spigot.nextDigit();
68                 handleDigit(digit);
69             }
70         }
71     };
72     producerThread.start();
73 }
74
75 /* pauseImpl will be implemented differently in a future subclass */
76 protected void pauseImpl() {
77     try {
78         Thread.sleep(Long.MAX_VALUE);
79     } catch (InterruptedException e) {}
80 }
81 private void play() {
82     paused = false;
83     playImpl();
84 }
85 /* playImpl will be implemented differently in a future subclass */
86 protected void playImpl() {
87     producerThread.interrupt();
88 }
89 private void pause() {
90     paused = true;
91 }
92
93 /* Play/Pause Button */
94 public void actionPerformed(ActionEvent e) {
95     String action = spigotButton.getText();
96     if ("Play".equals(action)) {
97         play();
98         spigotButton.setText("Pause");
99     } else {
100        pause();
101        spigotButton.setText("Play");
102    }
103 }
104
105 /* GUI-related methods */
106 protected void showLastLine() {
107     /*
108     * These swing component methods are not thread-safe
109     * so we use SwingUtilities.invokeLater
110     */
111     SwingUtilities.invokeLater(new Runnable() {
112         public void run() {
113             JScrollBar vScrollBar = scrollPane.getVerticalScrollBar();
114             int numLines = textArea.getLineCount();
115             int lineHeight =
116                 textArea.getScrollableUnitIncrement(
117                     scrollPane.getViewport().getViewRect(),
118                     SwingConstants.VERTICAL,
119                     -1);
120             int visibleHeight = vScrollBar.getVisibleAmount();
121             int numVisibleLines = visibleHeight / lineHeight;
122             int scroll = (numLines - numVisibleLines) * lineHeight;
123             vScrollBar.setValue(scroll);
124         }
125     });
126 }
127 protected void displayDigit(int digit) {
128     if (numDigits == 1) {
129         textArea.append(".");
130     }
131     textArea.append(String.valueOf(digit));
132     numDigits++;
133     if (numDigits > 1) {
134         if ((numDigits - 1) % 100 == 0) {
135             textArea.append("\n "); //JTextArea.append is thread-safe
136             showLastLine();
137         } else if ((numDigits - 1) % 5 == 0) {
138             textArea.append(" "); //JTextArea.append is thread-safe
139         }
140     }

```

```

141     }
142     /* handleDigit will be implemented differently in a future subclass */
143     protected void handleDigit(int digit) {
144         displayDigit(digit);
145     }
146     public static void main(String[] arg) {
147         JFrame f = new JFrame();
148         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
149         f.getContentPane().add(new PiPanel1(), BorderLayout.CENTER);
150         f.getContentPane().add(new chap16.clock.ClockPanel1(), BorderLayout.NORTH);
151         f.pack();
152         f.show();
153     }
154 }

```

16.9 chap16.pi.PiSpigot.java

```

1     package chap16.pi;
2
3     import java.math.BigInteger;
4
5     public class PiSpigot {
6         private static final BigInteger zero = BigInteger.ZERO;
7         private static final BigInteger one = BigInteger.ONE;
8         private static final BigInteger two = BigInteger.valueOf(2);
9         private static final BigInteger three = BigInteger.valueOf(3);
10        private static final BigInteger four = BigInteger.valueOf(4);
11        private static final BigInteger ten = BigInteger.valueOf(10);
12
13        private BigInteger q = one;
14        private BigInteger r = zero;
15        private BigInteger t = one;
16        private BigInteger k = one;
17        private BigInteger n = zero;
18
19        public int nextDigit() {
20            while (true) {
21                n = three.multiply(q).add(r).divide(t);
22                if (four.multiply(q).add(r).divide(t).equals(n)) {
23                    q = ten.multiply(q);
24                    r = ten.multiply(r.subtract(n.multiply(t)));
25                    return n.intValue();
26                } else {
27                    BigInteger temp_q = q.multiply(k);
28                    BigInteger temp_2k_plus_1 = two.multiply(k).add(one);
29                    BigInteger temp_r = two.multiply(q).add(r).multiply(temp_2k_plus_1);
30                    BigInteger temp_t = t.multiply(temp_2k_plus_1);
31                    BigInteger temp_k = k.add(one);
32                    q = temp_q;
33                    r = temp_r;
34                    t = temp_t;
35                    k = temp_k;
36                }
37            }
38        }
39    }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/pi/PiPanel1.java
java -cp classes chap16.pi.PiPanel1

```

After `PiPanel1`'s constructor assembles the GUI, it calls `startProducer()` (*line 55*) which creates a `Thread` whose `run()` method is an infinite digit-producing loop. Every time it goes through the body of the loop, it politely calls `Thread.yield()` (*line 63*). Calling `Thread.yield()` allows other threads that may have been waiting a chance to execute, possibly causing the current thread to pause its own execution. If no threads were waiting, the current thread just continues. Since digit producing is CPU intensive and because this loop never ends, calling `Thread.yield()` is more than just polite. It's the right thing to do. Table 16-6 lists `Thread`'s `yield()` method.

Method Name and Purpose
public static void yield() Allows other threads that may have been waiting a chance to execute, possibly causing the current thread to pause its own execution.

Table 16-6: Calling the Thread.yield() Method

Let's discuss the logic of the producer thread's run() method. (See lines 61 - 70 of example 16-8). After calling thread.yield(), the loop's next action depends on the value of the boolean variable, *paused*. If *paused* is false, the loop produces another digit, calls handleDigit() and repeats. If *paused* is true, the thread enters the while(paused) loop which calls *pauseImpl()* causing the thread to sleep. Unless it is interrupted, the producer thread will exit sleep(Long.MAX_VALUE) only after approximately 300 million years. If it does sleep this long, the while(paused) loop will notice that *paused* is still true and will send the thread back to sleep again. I think it's more likely that the producer thread will exit sleep by being interrupted. How about you?

If the "Play" button is pressed while the producer is sleeping, the play() method will set *paused* to false and call playImpl() which wakes the producerThread up by calling producerThread.interrupt(). This will cause the while(paused) loop to exit since *paused* is no longer true, and digit-producing to resume. If, on the other hand, the "Pause" button is pressed while the producer is running, the producer thread will enter the while(paused) loop automatically on the very next digit-producing iteration and begin sleeping. This use of sleep() and interrupt() is somewhat unconventional. Later in the chapter, we will discuss a more appropriate mechanism for achieving the same results.

Quick Review

The sleep() and interrupt() methods can be used as a mechanism for regulating a thread's use of the CPU. If a thread's run() method is time consuming, it should call yield() periodically to allow other threads a chance to run.

THREAD PRIORITY AND SCHEDULING

After all that has been said about multiple threads being able to run simultaneously, you should know that thread concurrency is not literally possible unless the machine running your program has multiple processors. If that is not the case, then concurrency is merely imitated by the JVM through a process called time scheduling where threads take turns using the CPU. The JVM determines which thread to run at any time through a priority-based scheduling algorithm. Threads have a priority that ranges in value from Thread.MIN_PRIORITY to Thread.MAX_PRIORITY. When a Thread is created, it is assigned the same priority as the thread that created it but it can be changed at any time later. A thread's actual priority is capped by its ThreadGroup's maximum priority which may be set through ThreadGroup.setMaxPriority(). Thread's and ThreadGroup's priority-related methods are listed in tables 16-7 and 16-8.

Method Name and Purpose
public final int getPriority() Gets this thread's priority.
public final void setPriority(int newPriority) Sets this thread's priority to the specified value or to its thread group's maximum priority – whichever is smaller.

Table 16-7: Thread's Priority-Related Methods

Method Name and Purpose
public final int getMaxPriority() Gets this thread group's maximum priority.
public final void setMaxPriority(int newMaxPriority) Sets this thread group's priority to the specified value or (if it has a parent thread group) to its parent's maximum priority – whichever is smaller.

Table 16-8: ThreadGroup's Priority-Related Methods

In order to accommodate many different platforms and operating systems, the Java programming language makes few guarantees regarding the scheduling of threads. The JVM chooses and schedules threads loosely according to the following algorithm:

1. If one thread has a higher priority than all other threads, that is the one it will run.
2. If more than one thread has the highest priority, one of them will be chosen to run.
3. A thread that is running will continue to run until either:
 - a) it voluntarily releases the CPU through `yield()` or `sleep()`
 - b) it is blocked waiting for a resource
 - c) it finishes its `run()` method
 - d) a thread with a higher priority becomes runnable

A thread is said to be *blocked* if its execution has paused to wait for a resource to become available. For instance, a thread that is writing to a file might block several times while the system performs some low-level disk management routines. A thread is said to have been *preempted* if the JVM stops running it in order to run a higher priority thread. As a consequence of the JVM's scheduling algorithm, if the currently running thread has equal or higher priority than its competitors and never blocks or voluntarily releases the CPU, it may prevent its competitors from running until it finishes. Such a thread is known as a “selfish” thread and the other threads are said to have “starved”. Thread starvation is generally not desirable. If a selfish thread has a higher priority than the AWT/Swing threads, for instance, then the user interface might actually appear to freeze. Therefore, it is incumbent upon the programmer to do his part to ensure fairness in scheduling. This is easily accomplished by inserting periodic calls to `yield()` (or `sleep()` when appropriate) within the run methods of long-running threads.

The next example takes two `PiPanel`s and puts them in a window with two `comboBox`s controlling the individual threads' priorities. Start them both with their `Play/Pause` buttons and watch them go. Adjust their priorities to view the effect on their own and each other's speeds.

16.10 chap16.priority.PriorityPiPanel.java

```

1      package chap16.priority;
2
3      import java.awt.BorderLayout;
4      import java.awt.GridLayout;
5      import java.awt.event.ActionEvent;
6      import java.awt.event.ActionListener;
7      import java.util.Vector;
8
9      import javax.swing.JComboBox;
10     import javax.swing.JFrame;
11
12     import chap16.pi.PiPanel;
13
14     public class PriorityPiPanel extends PiPanel {
15     public PriorityPiPanel() {
16         Vector choices = new Vector();
17         for (int i = Thread.MIN_PRIORITY; i <= Thread.MAX_PRIORITY; ++i) {
18             choices.add(new Integer(i));
19         }
20         final JComboBox cb = new JComboBox(choices);
21         cb.setMaximumRowCount(choices.size());
22         cb.setSelectedItem(new Integer(producerThread.getPriority()));
23         cb.addActionListener(new ActionListener() {
24             public void actionPerformed(ActionEvent e) {
25                 Integer item = (Integer)cb.getSelectedItem();
26                 int priority = item.intValue();
27                 producerThread.setPriority(priority);

```

```

28     }
29     });
30     buttonPanel.add(cb, BorderLayout.WEST);
31 }
32 public static void main(String[] arg) {
33     JFrame f = new JFrame();
34     f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
35     f.getContentPane().setLayout(new GridLayout(2, 1));
36     f.getContentPane().add(new PriorityPiPanel());
37     f.getContentPane().add(new PriorityPiPanel());
38     f.pack();
39     f.show();
40 }
41 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/priority/PriorityPiPanel.java
java -cp classes chap16.priority.PriorityPiPanel

```

YOUR RESULTS MAY VARY

There is variation in how systems handle multiple threads and thread priorities. Some systems employ a thread management scheme known as *time slicing* that prevents selfish threads from taking complete control over the CPU by forcing them to share CPU time. Some systems map the range of Thread priorities into a smaller range causing threads with close priorities to be treated equally. Some systems employ aging schemes or other variations of the basic algorithm to ensure, for instance, that even low-priority threads have a chance. Do not count on the underlying system's behavior for your program to run correctly. Program defensively (*and politely*) by using `Thread.sleep()`, `Thread.yield()` or your own mechanism for ensuring that your application threads share the CPU as intended. Do not write selfish threads unless there is a compelling need to do so.

Example 16.11 is intended to determine whether or not your system employs timeslicing. It creates two selfish maximum priority threads. Each of them maintains control over a variable that the other can see. Each of them monitors the other's variable for changes in value. While the current thread is running, if the variable controlled by the other thread changes value, then the current thread can conclude that it was preempted by the other thread. Being a selfish, highest priority thread, the only possible explanation for its preemption is that the underlying system was responsible. The program ends when and if one of the threads notices that it has been preempted. The longer the program runs without either thread being preempted, the greater the chance that the system does not employ time-slicing. If either of the threads completes its loop without being preempted, the program will conclude that the system doesn't employ time-slicing. It takes less than ten seconds on my computer for a thread to be preempted and the program to end. Figure 16-3 shows the console output from running example 16.11 on my computer.

16.11 chap16.timeslicing.SliceMeter.java

```

1     package chap16.timeslicing;
2
3     public class SliceMeter extends Thread {
4         private static int[] vals = new int[2];
5         private static boolean usesTimeSlicing = false;
6
7         private int myIndex;
8         private int otherIndex;
9
10        private SliceMeter(int myIndex, int otherIndex) {
11            this.myIndex = myIndex;
12            this.otherIndex = otherIndex;
13            setPriority(Thread.MAX_PRIORITY);
14        }
15        public void run() {
16            int lastOtherVal = vals[otherIndex];
17            for (int i = 1; i <= Integer.MAX_VALUE; ++i) {
18                if (usesTimeSlicing) {
19                    return;
20                }
21                int curOtherVal = vals[otherIndex];
22                if (curOtherVal != lastOtherVal) {
23                    usesTimeSlicing = true;
24                    int numLoops = curOtherVal - lastOtherVal;
25                    lastOtherVal = curOtherVal;
26                    System.out.println(
27                        ("While meter" + myIndex + " waited, "
28                         + ("meter" + otherIndex + " looped " + numLoops + " times"));

```



```

29     }
30     vals[myIndex] = i;
31 }
32 }
33 public static void main(String[] arg) {
34     SliceMeter meter0 = new SliceMeter(0, 1);
35     SliceMeter meter1 = new SliceMeter(1, 0);
36     meter0.start();
37     meter1.start();
38     try {
39         meter0.join();
40         meter1.join();
41     } catch (InterruptedException e) {
42         e.printStackTrace();
43     }
44     System.out.println("usesTimeSlicing = " + usesTimeSlicing);
45 }
46 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/timeslicing/SliceMeter.java
java -cp classes chap16.timeslicing.SliceMeter

```

```

While meter0 waited, meter1 looped 2648918 times
usesTimeSlicing = true

```

Figure 16-3: Results of Running Example 16.11

Incidentally, this program illustrates the use of the `join()` method. Without the calls to `join()` in lines 39 and 40, the main thread would report that `usesTimeSlicing` was false and terminate before `meter0` and `meter1` had completed their work. By calling `join()`, `SliceMeter` instructs the current thread, “main”, to wait for `meter0` and `meter1` to terminate before continuing its own execution. The syntax for `join()` can be confusing. The statement “`meter0.join()`”, for instance, doesn’t tell `meter0` to do anything. It tells the current thread to wait until `meter0` has terminated. Thread’s `join()` methods are listed in table 16-9.

Method Name and Purpose
public final void join() throws InterruptedException Causes the current thread to wait until this thread terminates.
public final void join(long millis) throws InterruptedException Causes the current thread to wait no longer than the specified time until this thread terminates.
public final void join(long millis, int nanos) throws InterruptedException Causes the current thread to wait no longer than the specified time until this thread terminates.

Table 16-9: Thread’s `join()` Methods

Quick Review

The JVM employs a preemptive algorithm for scheduling threads waiting to be run. Setting a thread’s priority can affect how and when the JVM schedules it, which, in the face of other threads competing for CPU time, can affect the thread’s performance. A thread should call `yield()` or `sleep()` or otherwise make provisions for sharing the CPU with competing threads. A thread that does not make provisions for sharing the CPU with competing threads (*by calling `yield()`, `sleep()` or some other mechanism*) is called a selfish thread. There is no guarantee that the underlying system will prevent selfish threads from taking complete control of the CPU. Because of variations in how systems handle the scheduling of threads, the correctness of a program’s behavior should not rely on the particulars of any system.

RACE CONDITIONS

Take a look at the following class:

16.12 chap16.race.LongSetter.java

```

1     package chap16.race;
2
3     public final class LongSetter {
4         private long x;
5
6         public boolean set(long xval) {
7             x = xval;
8             return (x == xval);
9         }
10    }
```

Knowing all that you know about Java programming at this point, answer the following question: Can an instance of LongSetter (*example 16.12*) ever be used in a program where its set() method might return false?

If your answer is yes, explain why. If your answer is no, explain why. Think carefully before going on.

I mean it. Don't read farther until you've thought about this. (*I know I can't really make you stop here and think this through but it's worth a try!*)

Do you have an answer? You do? Good!

The truth is that it is very simple to devise a multi-threaded program that causes the set() method to return false. If only one thread at a time ever calls LongSetter's set() method then all is fine, but if two or more threads call it at the same time, all bets are off. Example 16.13 is a program designed to "break" LongSetter. It creates four threads and starts them all hammering at the set() method of one LongSetter instance. As soon as the set method returns false, the program ends. Run it to see the results. Figure 16-4 shows the results of running example 16.13 on my computer.

16.13 chap16.race.Breaker.java

```

1     package chap16.race;
2
3     public class Breaker extends Thread {
4         private final static LongSetter longSetter = new LongSetter();
5
6         private final long value;
7
8         public Breaker(long value) {
9             this.value = value;
10        }
11        public void run() {
12            long count = 0;
13            boolean success;
14
15            while (true) {
16                success = longSetter.set(value);
17                count++;
18                if (!success) {
19                    System.out.println(
20                        "Breaker " + value + " broke after " + count + " tries.");
21                    System.exit(0);
22                }
23            }
24        }
25        public static void main(String[] arg) {
26            new Breaker(1).start();
27            new Breaker(2).start();
28            new Breaker(3).start();
29            new Breaker(4).start();
30        }
31    }
```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/race/Breaker.java
java -cp classes chap16.race.Breaker
```

Figure 16-4 shows the results of running this program.

Breaker 2 broke after 2554997 tries.

Figure 16-4: Results of Running Example 16.13

So what happened here? The flip side of thread independence is, unfortunately, thread non-cooperation. Left to its own devices, a thread charges recklessly forward, executing statement after statement in sequential order, caring not in the least what other reckless threads may be doing at the very same moment. And so it eventually happens in the Breaker program that after one thread executes the assignment “`x = val`” but before it executes the comparison “`return x == val`”, another thread tramples over the value of `x` by executing the assignment “`x = val`” with a different value for `val`. The diagram in figure 16-5 depicts just one of several possible unfortunate scenarios. Here, Breaker(3) sets `x` to 3 immediately after Breaker(2) sets it to 2. When it comes to the comparison, last one wins, so Breaker(2)’s comparison fails.

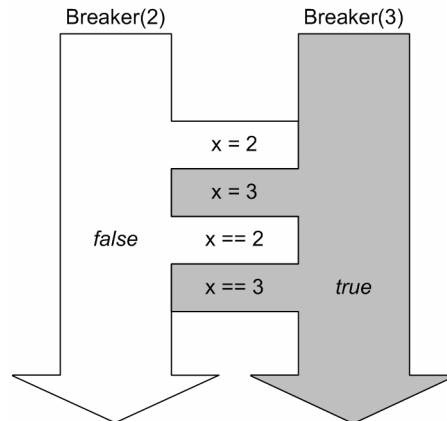


Figure 16-5: Breaker.java Thread Interaction

The implications of this are devastating. In a multithreaded application without proper defensive coding, a thread has no guarantee that the results of executing one statement will persist even until the beginning of the next statement’s execution. Actually, things are even worse than that because many single statements compile to more than one byte-code and threads can insert themselves between the execution of any two byte-codes. In this simple case, it is easy to see how thread interference negatively affected the results. But in an application of even moderate complexity, foreseeing the many different ways in which competing threads might interfere with each other can be very difficult if not virtually impossible.

The Breaker example above creates what is known as a *race condition*. This is a condition where more than one thread accesses the same resource at the same time, resulting in errors and/or resource corruption. You can think of the threads as racing with each other. In the case of Breaker, the last thread to set the value of `x` “wins” the race, but in other programs and situations, the first thread might “win”. Sometimes there is no “winner”. In any case, whether or not any particular thread wins, the program itself loses.

THE MECHANICS OF SYNCHRONIZATION

In order to defend a multi-threaded program from the ravages of unruly threads, it is necessary to delve deep into the Java language into the mechanics of Object itself, the root of all Java classes. Associated with every Object is a unique lock more formally known as a *monitor*. Every Object is said to “contain” this lock although no lock “attribute” is actually defined. Rather, the concept of a lock is embedded into the JVM implementation and is a part of Object’s low-level architecture. As such, it obeys different rules than do Objects and primitives. Here are some of the rules and behaviors of locks as they relate to threads:

1. A lock can be “owned” by a thread.
2. A thread becomes the owner of a lock by explicitly acquiring it.
3. A thread gives up ownership of a lock by explicitly releasing it.
4. If a thread attempts to acquire a lock not currently owned by another thread, it becomes the lock’s owner.
5. If a thread attempts to acquire a lock that is owned by another thread, then it blocks (*stops executing code*) at least until the lock is released by the current owner.
6. A thread may simultaneously own locks on any number of different Objects.

Figure 16-6 illustrates the process of a thread acquiring and releasing the lock of an Object, A.

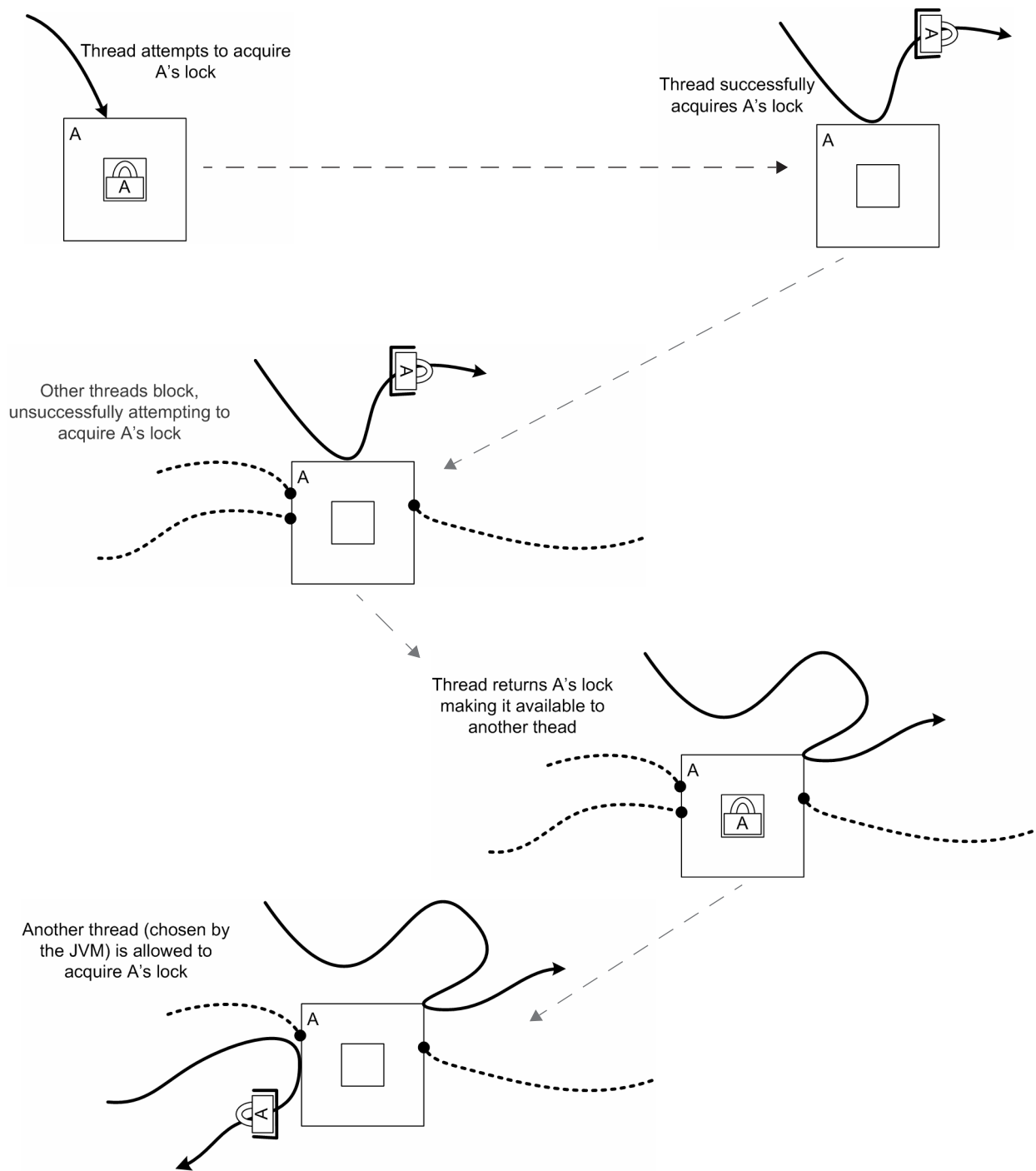


Figure 16-6: Acquiring and Releasing Locks

Although there are no actual method calls for acquiring or releasing a lock, there is a syntax for synchronizing a block of code on an object. A code block that is synchronized on an object can only be executed by a thread that owns that object's lock. There are no restrictions on what a synchronized code block may do. Here is an example of a block of code synchronized on "object":

```

/* Before entering, the current thread must either acquire or own object's lock */
synchronized (object) { //The current thread now owns object's lock

    /* Code placed here is synchronized on object's lock */

} // The current thread releases object's lock if it acquired it when entering

```

The `synchronized` statement requires a parameter of type `Object`. The `synchronized` code block is said to be synchronized “on” that object. When a thread that does not already own the object’s lock encounters a `synchronized` statement, it will attempt to acquire the lock. If the lock is not available, it will wait until the lock becomes available, at which point it will make another attempt to acquire the lock. When (*if ever*) it has successfully acquired the lock, it will enter and execute the `synchronized` block of code. When the thread finally exits the code block, the lock will be released. A thread that already owns the lock has no need to reacquire the lock when it encounters the `synchronized` block. It immediately enters and executes the code block. Nor will it release the lock upon exit. It will, of course, have to release the lock whenever it exits the block of code that initially caused it to acquire the lock.

Just as with any other code block, `synchronized` code blocks may be nested. This makes it possible for a thread to own more than one lock simultaneously. Due to this nested nature, a thread that owns more than one lock will always release its locks in the reverse order of the order in which they were acquired. Following is an example of a nested `synchronized` block of code.

```

synchronized (objectA) { //The current thread now owns objectA's lock

    synchronized (objectB) { //The current thread now owns objectB's lock also

        /* Code placed here is synchronized on objectA's and objectB's lock */

    } //The current thread releases objectB's lock if it acquired it when entering

} //The current thread releases objectA's lock if it acquired it when entering

```

THREE BASIC RULES OF SYNCHRONIZATION

There are three basic rules for utilizing synchronization effectively. We will explore them in this section.

Synchronization Rule 1

Synchronization can be used to ensure cooperation between threads competing for common resources.

Let’s see in a diagram similar to figure 16-5 what might happen if we put the call to `LongSetter.set()` inside a code block that is synchronized on an `Object`, A. This would require all threads to own or acquire a lock on A before executing the two statements of `LongSetter.set()`. In figure 16-7, we’ll have `Breaker(3)` attempt to execute the method after `Breaker(2)` has begun. This is the same scenario as figure 16-5 depicted but this time, as figure 16-7 shows, `Breaker(3)` is forced to block because `Breaker(2)` owns the lock. `Breaker(2)` finishes the code block and releases the lock. Then `Breaker(3)` attempts to acquire the lock, succeeds and executes the code block. This illustrates how the use of synchronization can prevent race conditions from occurring.

Let’s test this approach with some code. `SynchedBreaker` (*example 16.14*) synchronizes on the `LongSetter` instance. Since it is a static field, all instances of `SynchedBreaker` will synchronize on the same object.

16.14 *chap16.race.SynchedBreaker.java*

```

1     package chap16.race;
2
3     public class SynchedBreaker extends Thread {
4         private final static LongSetter longSetter = new LongSetter();
5
6         private final long value;
7
8         public SynchedBreaker(long value) {
9             this.value = value;
10        }
11        public void run() {
12            long count = 0;
13            boolean success;
14
15            while (true) {

```

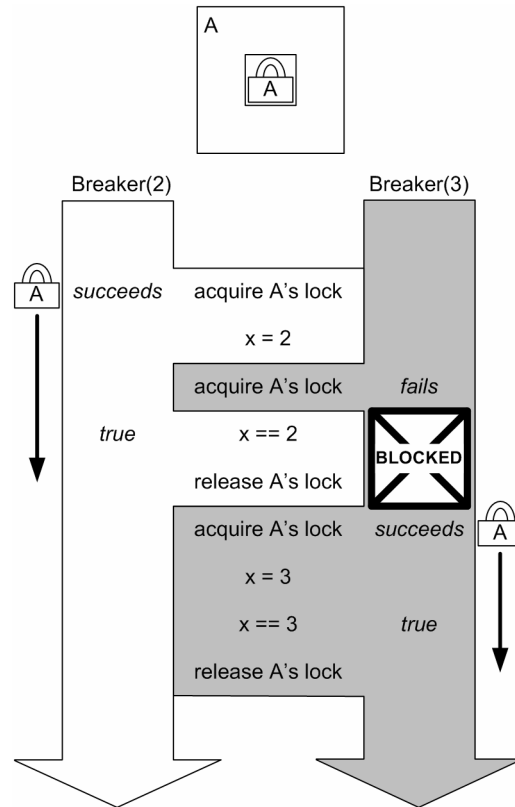


Figure 16-7: Breaker(2) and Breaker(3) Both Succeed

```

16     synchronized (longSetter) {
17         success = longSetter.set(value);
18     }
19     count++;
20     if (!success) {
21         System.out.println(
22             "Breaker " + value + " broke after " + count + " tries.");
23         System.exit(0);
24     }
25 }
26 }
27 public static void main(String[] arg) {
28     new SynchedBreaker(1).start();
29     new SynchedBreaker(2).start();
30     new SynchedBreaker(3).start();
31     new SynchedBreaker(4).start();
32 }
33 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/race/SynchedBreaker.java
java -cp classes chap16.race.SynchedBreaker

```

As you can see from running it, SynchedBreaker doesn't break. When you are satisfied that it will never break, go ahead and terminate it.

SYNCHRONIZATION RULE 2

All threads competing for common resources must synchronize their access or none is safe.

In figures 16-8 and 16-9, we show what might happen if some but not all competing threads synchronize their access to shared resources. Here, Breaker(2) synchronizes its access but Breaker(3) does not. Consequently, Breaker(2)'s acquisition of A's lock does nothing to prevent Breaker(3) from unsynchronized access. Nor does it protect Breaker(3) from Breaker(2)'s intrusion.

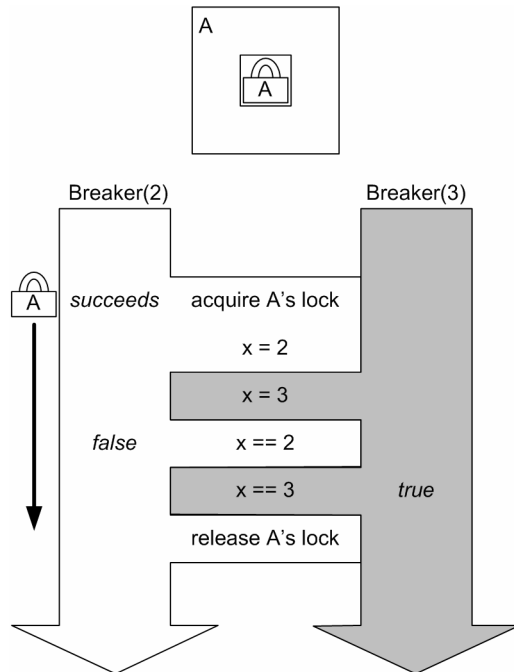


Figure 16-8: Breaker(2) Fails Because Not All Threads Synchronized

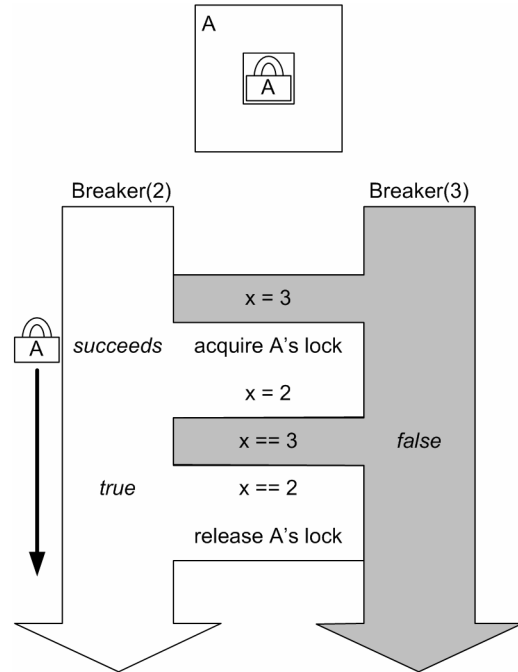


Figure 16-9: Breaker(3) Fails Because Not All Threads Synchronized

This rule is verified by `MixedModeBreaker` (example 16.15), where one thread synchronizes on the `LongSetter` instance before invoking the `set()` method and three threads don't. Figure 16-10 shows the results of running example 16.15.

16.15 chap16.race.MixedModeBreaker.java

```

1  package chap16.race;
2
3  public class MixedModeBreaker extends Thread {
4      private final static LongSetter longSetter = new LongSetter();
5
6      private final long value;
7      private boolean synch;
8
9      public MixedModeBreaker(long value, boolean synch) {
10         this.value = value;
11         this.synch = synch;
12     }
13     public void run() {
14         long count = 0;
15         boolean success;
16
17         while (true) {
18             if (synch) {
19                 synchronized (longSetter) {
20                     success = longSetter.set(value);
21                 }
22             } else {
23                 success = longSetter.set(value);
24             }
25             count++;
26             if (synch && !success) {
27                 System.out.println(
28                     "Breaker " + value + " broke after " + count + " tries.");
29                 System.exit(0);
30             }
31         }
32     }
33     public static void main(String[] arg) {
34         new MixedModeBreaker(1, false).start();
35         new MixedModeBreaker(2, false).start();
36         new MixedModeBreaker(3, false).start();
37         new MixedModeBreaker(4, true).start();
38     }
39 }

```

Use the following command to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/race/MixedModeBreaker.java
java -cp classes chap16.race.MixedModeBreaker
```

Breaker 4 broke after 2648189 tries.

Figure 16-10: Results of Running Example 16.15

SYNCHRONIZATION RULE 3

All threads competing for common resources must synchronize their access on the same lock or none is safe.

In Figure 16-11, Breaker(2) and Breaker(3) both have synchronized access to `LongSetter.set()` but they synchronize on different objects, Breaker(2) on A and Breaker(3) on B. Again, the use of synchronization here is ineffective. Breaker(2)'s acquisition of A's lock has no effect on Breaker(3)'s ability to acquire B's lock and visa versa.

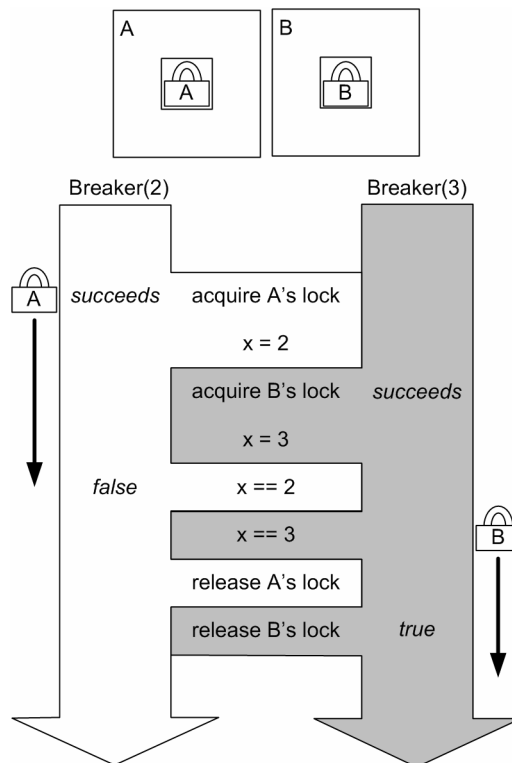


Figure 16-11: Breaker(2) Fails Because Threads Synchronized on Different Locks

We verify rule 3 with `PointlessSynchedBreaker` (example 16.16) where each `PointlessSynchedBreaker` synchronizes on itself rather than on a common object. Figure 16-12 shows the results of running example 16.16.

16.16 `chap16.race.PointlessSynchedBreaker.java`

```
1 package chap16.race;
2
3 public class PointlessSynchedBreaker extends Thread {
4     private final static LongSetter longSetter = new LongSetter();
5
6     private final long value;
7
8     public PointlessSynchedBreaker(long value) {
9         this.value = value;
10    }
11    public void run() {
```



```

12     long count = 0;
13     boolean success;
14
15     while (true) {
16         synchronized (this) {
17             success = longSetter.set(value);
18         }
19         count++;
20         if (!success) {
21             System.out.println(
22                 "Breaker " + value + " broke after " + count + " tries.");
23             System.exit(0);
24         }
25     }
26 }
27 public static void main(String[] arg) {
28     new PointlessSynchedBreaker(1).start();
29     new PointlessSynchedBreaker(2).start();
30     new PointlessSynchedBreaker(3).start();
31     new PointlessSynchedBreaker(4).start();
32 }
33 }

```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/race/PointlessSynchedBreaker.java
java -cp classes chap16.race.PointlessSynchedBreaker

```

```

Breaker 1 broke after 37265504 tries.

```

Figure 16-12: Results of Running Example 16.16

SYNCHRONIZING METHODS

What if we didn't want or weren't able to modify Breaker's code in order to prevent race conditions with LongSetter? Is there something else we could do? Yes, we could modify LongSetter itself as in the following code for SynchedLongSetter.

16.17 chap16.race.SynchedLongSetter

```

1     package chap16.race;
2
3     public final class SynchedLongSetter {
4         private long x;
5
6         public boolean set(long xval) {
7             synchronized (this) {
8                 x = xval;
9                 return (x == xval);
10            }
11        }
12    }

```

From the point of view of execution, using Breaker with SynchedLongSetter is identical to using SynchedBreaker with LongSetter. In each approach, the assignment and comparison statements within the set() method are protected by requiring the current thread to own the LongSetter (or SynchedLongSetter) instance's lock. In the first approach, SynchedBreaker assumes responsibility for synchronization; in the second, SynchedLongSetter assumes responsibility. The pattern of synchronizing an entire method on "this" is so common place that there is an alternate notation for it. The following two methods, for instance, are equivalent. Before allowing a thread to execute the method body, they both require it to own (or acquire) the lock of the Object instance whose method is being invoked. Methods declared in this shorthand way are called *synchronized methods*.

```

public class Example {
    public synchronized void doSomething() {
        /* code block */
    }
}

```

is shorthand for

```
public class Example {
    public void doSomething () {
        synchronized (this) {
            /* code block */
        }
    }
}
```

This alternate notation applies to static methods as well. The following two static methods are equivalent to each other. Before allowing a thread to execute the method body, they both require it to own (*or acquire*) the lock of the Class object associated with the class that defines the method.

```
public class Example {
    public synchronized static void doSomethingStatic () {
        /* code block */
    }
}
```

is shorthand for

```
public class Example {
    public static void doSomethingStatic () {
        synchronized (Example.class) {
            /* code block */
        }
    }
}
```

Please keep in mind that these are simply alternate notations. No matter how it may look, the word “synchronized” is not part of a method’s signature and it is not inherited by subclasses. In the following two classes, for example, `Example.doSomething()` is synchronized but `Example2.doSomething()` is not.

```
public class Example {
    public synchronized void doSomething () {
        /* code block */
    }
}

public class Example2 extends Example {
    public void doSomething () {
        /* code block */
    }
}
```

Quick Review

A race condition is a condition where more than one thread accesses the same resource at the same time, resulting in errors and/or resource corruption. Synchronization can be used to prevent race conditions by ensuring that a block of code is executed by only one thread at a time. Synchronization is only effective when all threads attempting to execute a block of code are synchronized on a common object. Synchronization is applied to a block of code. Synchronized methods have a slightly different syntax but they too are simply blocks of code that have been synchronized.

THE PRODUCER-CONSUMER RELATIONSHIP

There are cases where two or more running threads share resources and also depend on each other to update the resources. This is typified in the *producer-consumer* relationship where a producer thread produces a resource that a consumer thread consumes. Synchronizing each thread's access to the resource can prevent race conditions but some mechanism is still needed for the threads to communicate with each other. Looking at it from the consumer's point of view, if the consumer needs to consume a resource that the producer has not yet produced, then the consumer must wait until it is notified that the resource has been produced. Looking at it from the producer's point of view, if storage is limited and the producer has temporarily stopped producing because there is no more room, then the producer will need to be notified when the consumer has consumed some of the storage.

In `PiPanel1` (*example 16.8*), as you may remember, a producer thread kept producing digits and calling `handleDigit()` every time a new digit was produced. The `handleDigit()` method consumed the digits by calling `displayDigit()` which displayed them in a `JTextArea`. This was not a producer-consumer relationship because the call to `displayDigit()` was issued from the producer thread itself. In other words, the producer was its own consumer, making it a logical impossibility for it to produce a digit unless the previous digit had been consumed.

In `PiPanelProdCons` (*example 16.20*), we will separate out the consumer functionality into a separate thread to create a true producer-consumer relationship. First, take a look at `DefaultDigitHolder.java` (*example 16.19*) which implements `DigitHolder` (*example 16.18*). `PiPanelProdCons` will use an instance of `DefaultDigitHolder` as the storage facility for a single digit. The producer thread will store digits into it and the consumer thread will retrieve digits from it. Notice that whenever `store()` is called, the previously stored digit value will be overwritten by the new value. `DefaultDigitHolder` has a storage capacity of one digit.

16.18 *chap16.prodcons.DigitHolder*

```
1 package chap16.prodcons;
2
3 public interface DigitHolder {
4     public void store(int digit);
5     public int retrieve();
6 }
```

16.19 *chap16.prodcons.DefaultDigitHolder.java*

```
1 package chap16.prodcons;
2
3 public class DefaultDigitHolder implements DigitHolder {
4     private int digit;
5
6     public void store(int digit) {
7         this.digit = digit;
8     }
9     public int retrieve() {
10         return digit;
11     }
12 }
```

`PiPanelProdCons` extends `PiPanel1`. It overrides `handleDigit()` to store the digit in a `DigitHolder` thereby removing the consumer functionality from the producer thread. `PiPanelProdCons`'s constructor creates and starts a consumer thread that continuously (*and rapidly*) retrieves digits from the `DigitHolder` and calls `displayDigit()`. The `displayDigit()` method, as we remember from `PiPanel1`, appends to the `JTextArea` whatever digit the consumer gives it. This is a true consumer-producer relationship in that the producer keeps producing and the consumer keeps consuming. Unfortunately, there is no coordination between the two.

`PiPanelProdCons` can run in three modes: "default", "good" and "bad" depending on its command line arguments. These modes specify whether to use an instance of `DefaultDigitHolder` itself or a particular subclass of it. The absence of command-line arguments will cause `PiPanelProdCons` to use an instance of `DefaultDigitHolder`.

16.20 *chap16.prodcons.PiPanelProdCons.java*

```
1 package chap16.prodcons;
2
3 import java.awt.BorderLayout;
4
5 import javax.swing.JFrame;
6
7 import chap16.pi.PiPanel1;
8
9 public class PiPanelProdCons extends PiPanel1 {
```

```

10     private DigitHolder holder;
11
12     public PiPanelProdCons(DigitHolder holder) {
13         this.holder = holder;
14         startConsumer();
15     }
16     private void startConsumer() {
17         Thread consumerThread = new Thread() {
18             public void run() {
19                 while (true) {
20                     int digit = holder.retrieve();
21                     displayDigit(digit);
22                 }
23             }
24         };
25         consumerThread.start();
26     }
27     //called constantly by producer thread
28     protected void handleDigit(int digit) {
29         holder.store(digit);
30     }
31     public static void main(String[] arg) {
32         JFrame f = new JFrame();
33         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
34         DigitHolder holder = null;
35         if (arg.length > 0) {
36             if ("bad".equals(arg[0])) {
37                 holder = new BadDigitHolder();
38             } else if ("good".equals(arg[0])) {
39                 holder = new GoodDigitHolder();
40             }
41         }
42         if (holder == null) {
43             holder = new DefaultDigitHolder();
44         }
45         f.getContentPane().add(new PiPanelProdCons(holder), BorderLayout.CENTER);
46         f.getContentPane().add(new chap16.clock.ClockPanel1(), BorderLayout.NORTH);
47         f.pack();
48         f.show();
49     }
50 }

```

Try to predict what will happen when the program uses a `DefaultDigitHolder`. Remember that the consumer thread will be started by `PiPanelProdCons`'s constructor so it will be running before the press of the "Play" button has even started the producer thread. When you think you know what will happen, run the program and find out. Make sure you press the Pause/Play button at least once to start the producer thread.

Use the following commands to compile and run the examples. From the directory containing the `src` folder:

```

javac -d classes -sourcepath src src/chap16/prodcons/PiPanelProdCons.java
java -cp classes chap16.prodcons.PiPanelProdCons

```

Were you surprised by the results? Since it's a lot quicker to consume a digit of pi than it is to produce it, the consumer thread is able to get and display the most recent digit many times before the producer thread is able to produce the next digit. Running `PiPanelProdCons` using a `DefaultDigitHolder` simply highlights the need to coordinate the producer and consumer threads' activities.

Example 16.21, `BadDigitHolder.java` tries, unsuccessfully, to fix the problem by synchronizing `DefaultDigitHolder`'s store and retrieve methods. The synchronization is successful, as far as it goes, in preventing more than one thread from storing simultaneously or retrieving simultaneously. Furthermore, because both methods synchronize on the same object, it even prevents one thread from storing while another is retrieving. Unfortunately, it does nothing further to coordinate the producer and consumer threads' activities. Run the program again with a command line argument of "bad" to use `BadDigitHolder` and to verify that synchronization alone will not be enough in this case.

16.21 chap16.prodcons.BadDigitHolder.java

```

1     package chap16.prodcons;
2
3     public class BadDigitHolder extends DefaultDigitHolder {
4         public synchronized void store(int digit) {
5             super.store(digit);
6         }
7         public synchronized int retrieve() {
8             return super.retrieve();
9         }
10    }

```

Use the following command to execute the example. From the directory containing the src folder:

```
java -cp classes chap16.prodcons.PiPanelProdCons bad
```

The real problem to be solved is that the consumer thread needs to be notified when the producer thread has produced a digit, and the producer thread needs to be notified when the consumer thread has consumed a digit. The Object class itself defines five methods that can be used to achieve an inter-thread notification mechanism. These methods, listed in table 16-10, are useful when threads need to wait on conditions that are set by other threads.

Method Name and Purpose
public final void wait() throws InterruptedException Causes the current thread to wait for another thread to call notify() or notifyAll() on this same object.
public final void wait(long timeout) throws InterruptedException Causes the current thread to wait no longer than the specified time for another thread to call notify() or notifyAll() on this same object.
public final void wait(long timeout, int nanos) throws InterruptedException Causes the current thread to wait no longer than the specified time for another thread to call notify() or notifyAll() on this same object.
public final void notify() Wakes up one thread (chosen by the JVM) that is waiting for this object's lock.
public final void notifyAll() Wakes up all threads that are waiting for this object's lock. They will still have to compete as usual for ownership of the lock.

Table 16-10: Object's Wait() and Notify() Methods

A thread that attempts to invoke these methods on an object will throw an `IllegalMonitorStateException` unless that thread owns the object's lock. (*Being a subclass of `RuntimeException`, `IllegalMonitorStateException` need not be declared or caught.*) Invoking `wait()` on an object causes the current thread to give up ownership of the object's lock and to suspend execution until another thread invokes either `interrupt()` on the waiting thread or `notify()` or `notifyAll()` on the object for which the thread is waiting. If a thread invokes the `notifyAll()` method on an object, then all threads that are waiting for that object are woken. If a thread invokes `notify()` on an object, the scheduler chooses just one thread to wake up from among the threads waiting for that object. Waiting threads are woken with an `InterruptedException` which they may handle however they choose. In the case of the timed `wait()` methods, a waiting thread will automatically exit the wait state if the specified time passes without a notification or interruption.

In example 16.22, `GoodDigitHolder` solves the producer-consumer coordination through the use of the `wait()` and `notify()` methods in conjunction with the boolean field, `haveDigit`, which indicates whether a digit was most recently retrieved or stored. If a digit was most recently stored, then `haveDigit` will be true, meaning that a digit is available for consumption. If a digit was most recently retrieved, then `haveDigit` will be false, meaning that a new digit will need to be stored before one is available for consumption. Because of the way the while loops are constructed, a thread that is waiting to store a digit will repeatedly wait until `haveDigit` is false. Conversely, a thread that is waiting to retrieve a digit will repeatedly wait until `haveDigit` is true.

Notice in lines 9 and 14 that the object upon which `GoodDigitHolder` invokes `wait()` and `notify()` is the `GoodDigitHolder` instance itself. Since these invocations occur within methods that are synchronized on the `GoodDigitHolder` instance, an `IllegalMonitorStateException` will not be thrown.

16.22 *chap16.prodcons.GoodDigitHolder*

```
1     package chap16.prodcons;
2
3     public class GoodDigitHolder extends DefaultDigitHolder {
4         private boolean haveDigit = false;
5
6         public synchronized void store(int digit) {
7             while (haveDigit) {
8                 try {
9                     wait();
10                } catch (InterruptedException e) {}

```

```

11     }
12     super.store(digit);
13     haveDigit = true;
14     notify();
15 }
16 public synchronized int retrieve() {
17     while (!haveDigit) {
18         try {
19             wait();
20         } catch (InterruptedException e) {}
21     }
22     haveDigit = false;
23     notify();
24     return super.retrieve();
25 }
26 }

```

Use the following command to execute the example. From the directory containing the src folder:

```
java -cp classes chap16.prodcons.PiPanel2 good
```

If this all seems clear to you, great. But in case it isn't, we'll look at what's happening in greater detail. First, we'll look at things from the consumer's point of view. If the consumer thread calls `retrieve()` when `haveDigit` is false, it means that the producer has not yet stored another digit. The consumer will enter the wait state from which it will not exit until it is either notified or interrupted. When it does exit `wait()`, it will have reacquired the lock and it will again check the loop condition (*possibly causing it to repeat the waiting process*). Consequently, the consumer thread will not exit the while loop unless `haveDigit` is true (*meaning that a new digit has been produced*). When, if ever, `haveDigit` becomes true, it will set `haveDigit` to false, the stored digit will be returned and any threads waiting for the lock will be notified that this might be a good time to wake up and try again. In particular, if the producer thread is waiting, it will wake up and discover that it is now safe to store another digit. Figure 16-13 shows what might happen if `haveDigit` is false when the consumer thread calls `retrieve()`.

Now let's look at things from the producer's point of view. If the producer thread calls `store()` when `haveDigit` is true, it means that the consumer has not yet retrieved the most recently stored digit. The producer will enter the wait state from which it will not exit until it is either notified or interrupted. When it does exit `wait()`, it will have reacquired the lock and it will again check the loop condition (*possibly causing it to repeat the waiting process*). Consequently, the producer thread will not exit the while loop unless `haveDigit` is false (*meaning that the most recently stored digit has been consumed*). When, if ever, `haveDigit` becomes false, it will set `haveDigit` to true, the newly produced digit will be stored and any threads waiting for the lock will be notified that this might be a good time to wake up and try again. In particular, if the consumer thread is waiting, it will wake up and discover that it is now safe to retrieve a digit. Figure 16-14 shows what might happen if `haveDigit` is true when the producer thread calls `store()`.

Earlier in the chapter in `PiPanel1` (*example 16.8*), the Swing event thread needed to notify a paused producer thread whenever the user pressed the "Play" button. `PiPanel1` used `sleep()` and `interrupt()` to effect this notification, but `wait()` and `notify()` are more appropriate. `PiPanel2` (*example 16.22*) extends `PiPanel1` solely to re-implement `pauseImpl()` and `playImpl()` using `wait()` and `notify()`. In order for `wait()` and `notify()` to work together, a single object on which to synchronize needs to be chosen. Any object including the `PiPanel2` instance itself could work, but in this case using the `PiPanel2` instance wouldn't be a good idea. If `PiPanel1` were ever modified to perform some synchronization on itself, there could be unintended implications on previously written subclasses (*such as PiPanel2*) that were also using "this" for their own synchronization. For this reason, we choose to synchronize `pauseImpl()` and `playImpl()` on an object known only to `PiPanel2`.

16.23 `chap16.prodcons.PiPanel2.java`

```

1     package chap16.prodcons;
2
3     import chap16.pi.PiPanel1;
4
5     public class PiPanel2 extends PiPanel1 {
6         private Object lock = new Object();
7
8         protected void pauseImpl() {
9             synchronized (lock) {
10                 try {
11                     lock.wait();
12                 } catch (InterruptedException e) {}
13             }
14         }
15         protected void playImpl() {

```

```

16     synchronized (lock) {
17         lock.notify();
18     }
19 }
20 }
    
```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```

javac -d classes -sourcepath src src/chap16/prodcons/PiPanel2.java
java -cp classes chap16.prodcons.PiPanel2
    
```

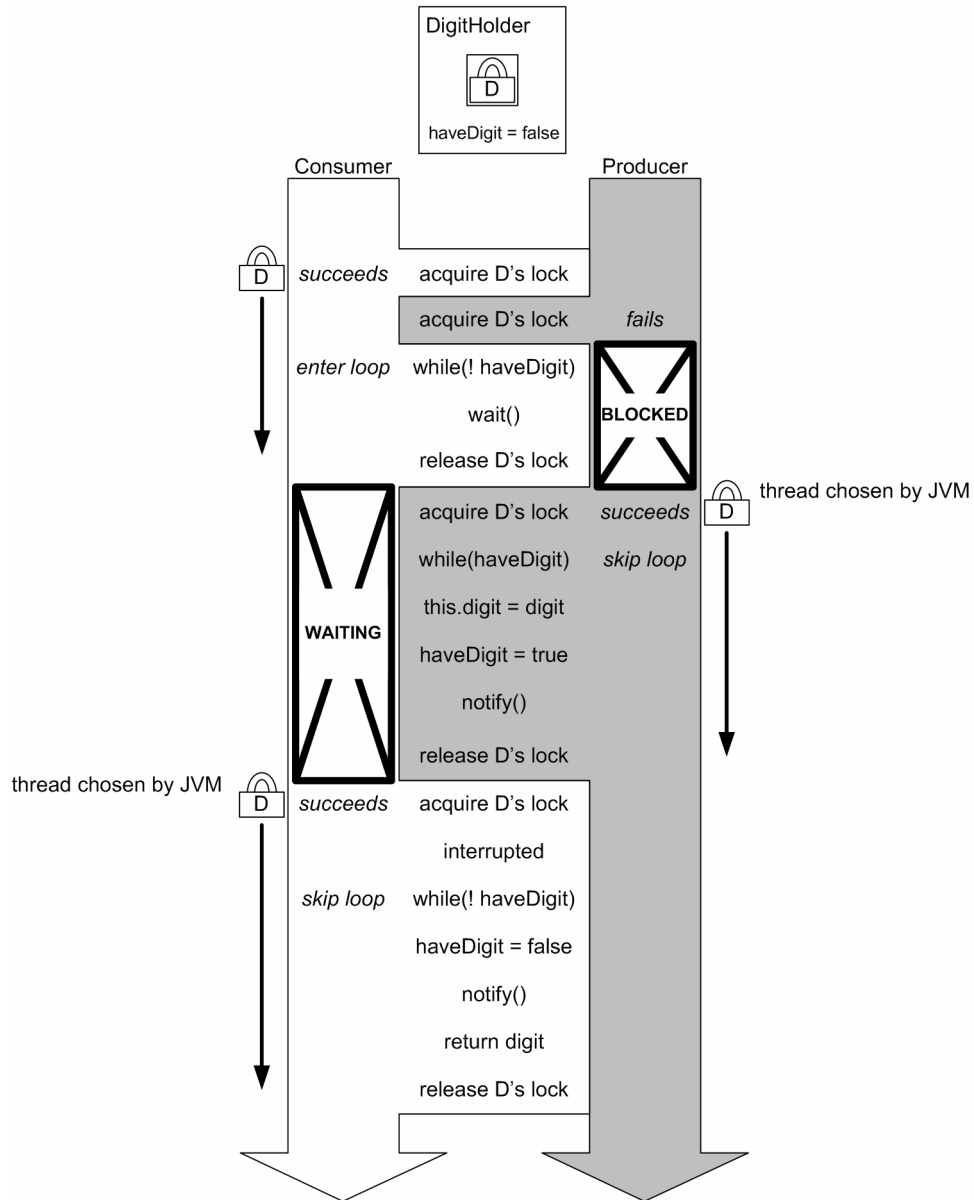


Figure 16-13: Consumer Thread Waits

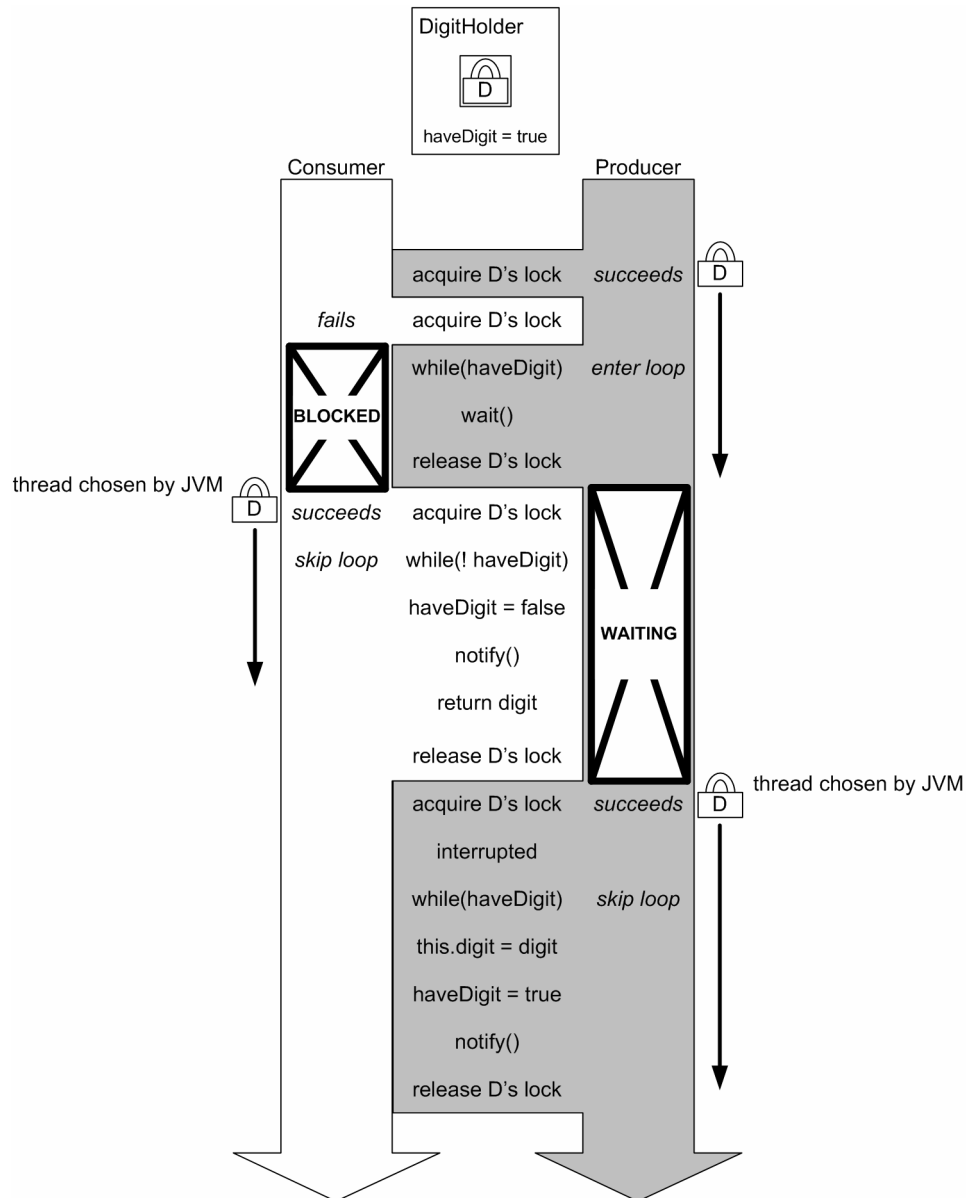


Figure 16-14: Producer Thread Waits

Quick Review

In a producer-consumer relationship a producer thread produces something that a consumer thread consumes and neither thread makes an effort to coordinate with the other. Coordination is handled instead by the object they both access. The `wait()` and `notify()` methods of `Object` can be used to negotiate the coordination between inter-dependent threads.

DEADLOCK

Synchronization saves the day when it comes to thread race conditions, and it is indispensable for enabling inter-thread notification but it does come at a price. There is some overhead and loss of speed associated with the acquiring and releasing of locks. And what about the fact that blocked threads make no forward progress? That is why, for instance, when Java 1.2 introduced the Collections framework, they chose to not make any of their Collection classes thread-safe even though the previous Java 1.0 collection-type classes were thread-safe. Why pay the price of synchronization when most applications won't need it? A multi-threaded application can always synchronize as necessary to accommodate a thread-unsafe class just as SynchedBreaker safely managed the thread-unsafe LongSetter.

There is another price that synchronization brings with it, and that is the danger of deadlock which can occur in certain circumstances when threads can hold more than one lock at the same time. Deadlock is the situation where two or more threads remain permanently blocked by each other because they are waiting for locks that each other already holds. Without some sort of tie-breaking mechanism, the threads block forever, resources are tied up and the application loses whatever functionality the deadlocked threads had provided. In figure 16-15, while a thread holding A's lock is blocked waiting to acquire B's lock, another thread holding B's lock is blocked waiting to acquire A's lock.

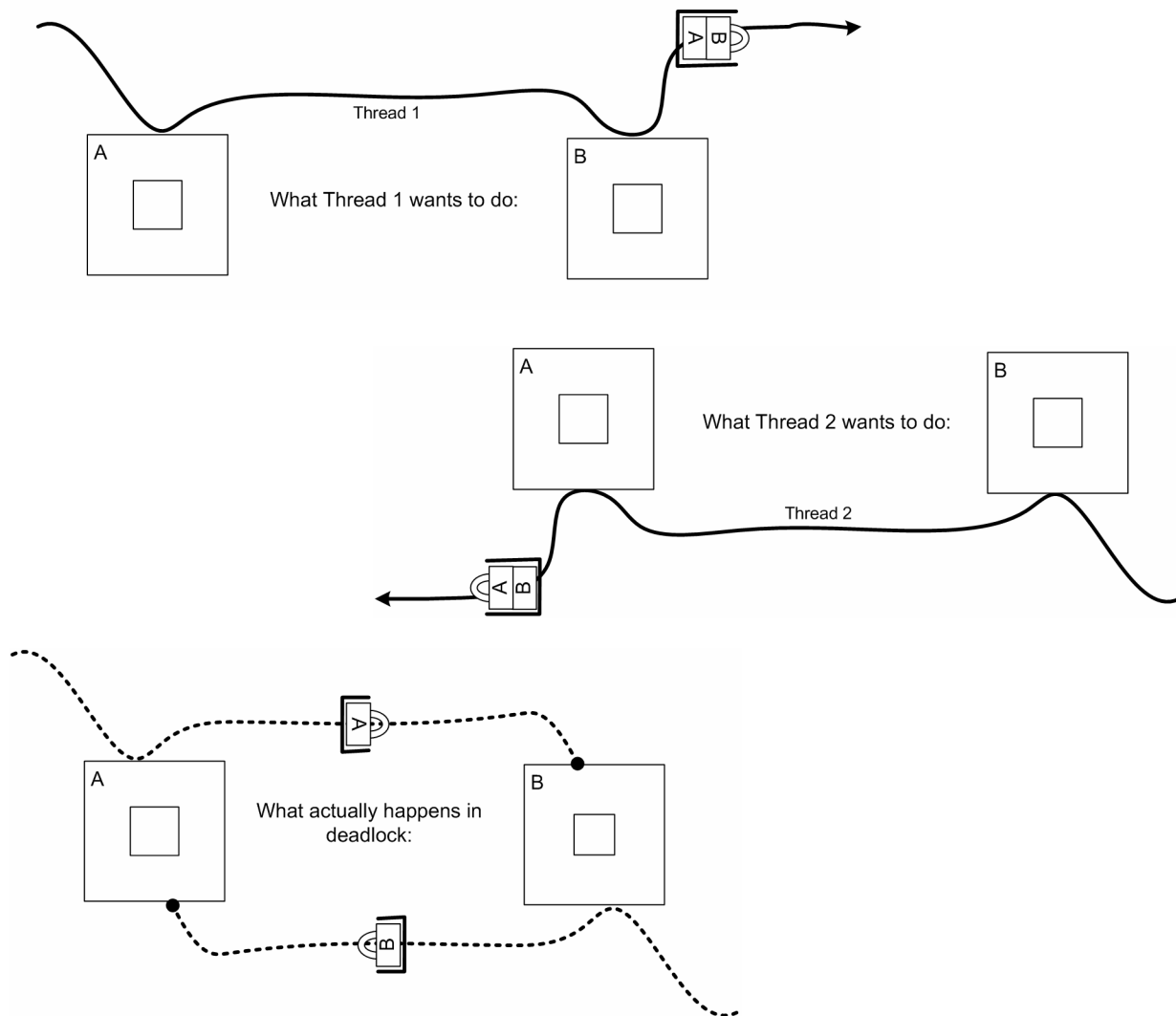


Figure 16-15: Deadlocked Threads

The following program creates a real-life example of deadlock. Be forewarned that you will have to forcibly terminate this program. Consider the Caller class (*example 16.24*) and the Breaker program (*example 16.25*) that uses it. At first glance there may not appear to be any nested synchronization but there is. The `answer()` and `call()` methods are both synchronized on the Caller instance. When a thread invokes the `call()` method on a Caller instance the current thread must acquire that Caller's lock. Since the `call()` method invokes the `answer()` method on another Caller instance, the thread must acquire the other Caller's lock also. Therefore, in order for a thread to completely execute the `call()` method of a Caller instance, there is a moment when it must own the locks of both Callers. This vulnerability is cruelly exploited by the Breaker program. The program runs smoothly as long as one caller calls the other and the other answers right away. But it will freeze if at some point the caller calls and, before the would-be answerer has a chance to answer, it calls the caller. At this point, the program will have achieved deadlock.

16.24 *chap16.deadlock.Caller.java*

```

1      package chap16.deadlock;
2
3      public class Caller {
4          private String name;
5
6          public Caller(String s) {
7              name = s;
8          }
9          public synchronized void answer() {
10             System.out.println(name + " is available!");
11         }
12         public synchronized void call(Caller other) {
13             System.out.println(this + " calling " + other);
14             other.answer();
15         }
16         public String toString() {
17             return name;
18         }
19     }

```

16.25 *chap16.deadlock.Breaker.java*

```

1      package chap16.deadlock;
2
3      public class Breaker extends Thread {
4          private Caller caller;
5          private Caller answerer;
6
7          public Breaker(Caller caller, Caller answerer) {
8              this.caller = caller;
9              this.answerer = answerer;
10         }
11         public void run() {
12             while (true) {
13                 caller.call(answerer);
14             }
15         }
16         public static void main(String[] arg) {
17             Caller a = new Caller("A");
18             Caller b = new Caller("B");
19
20             Breaker breakerA = new Breaker(a, b);
21             Breaker breakerB = new Breaker(b, a);
22
23             breakerA.start();
24             breakerB.start();
25         }
26     }

```

Use the following commands to compile and execute the example. From the directory containing the `src` folder:

```

javac -d classes -sourcepath src src/chap16/deadlock/Breaker.java
java -cp classes chap16.deadlock.Breaker

```

Figure 16-16 shows the output to the console from running example 16.24. Figure 16-17 illustrates what happens when the two threads enter the deadlock state.

Unfortunately, the Java language provides no facility for detecting or freeing deadlocked threads beyond shutting the JVM down. The best way to fix deadlock is to avoid it, so use extreme caution when writing multi-threaded programs where threads can hold multiple locks. Do not synchronize more than necessary and understand the various ways the threads you write might interact with each other. Keep the use of synchronization to a minimum and be especially careful about situations where a thread can hold multiple locks.

```

A calling B
B is available!
A calling B
B is available!
[this continues for a while until]...
B calling A
A is available!
B calling A
A is available!
[this continues for a while until]...
A calling B
B calling A
[deadlock]
    
```

Figure 16-16: Results of Running Example 16.25
(Output edited and annotated for brevity.)

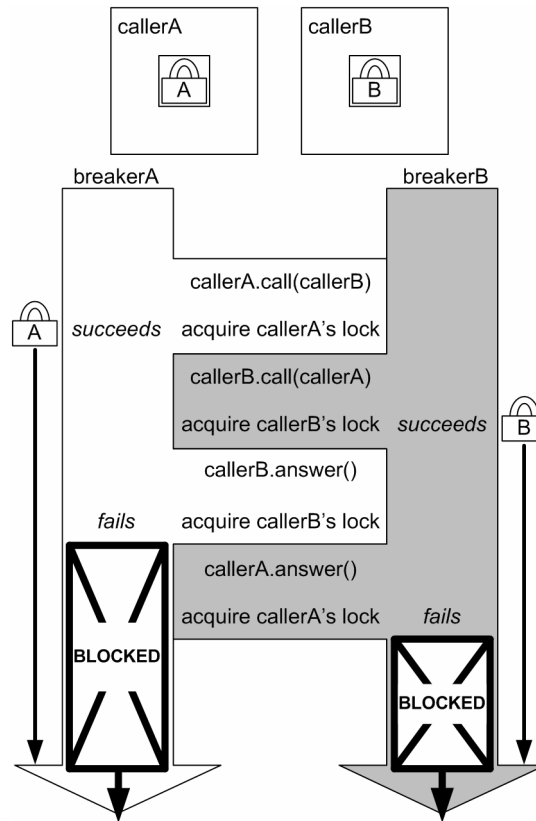


Figure 16-17: Deadlock Due to Nested Synchronization

Quick Review

Deadlock is the situation where two or more threads remain permanently blocked because they are waiting for locks each other already holds. Deadlock should be avoided and cannot be easily detected. The only sure method for

avoiding deadlock is to understand the threads you write and their potential interactions, and to program in such a way that deadlock is never a possibility.

ABOUT THE PI-GENERATING ALGORITHM

The algorithm used for the calculation of pi in this chapter is an example of a *spigot algorithm*. A spigot algorithm outputs digits incrementally, and does not reuse them after producing them. Spigot algorithms are known for both pi and e. A spigot algorithm for calculating the value of pi to any predetermined number of digits was first published by Rabinowitz and Wagon in 1995. It is not limited in the number of digits it can produce, but it is inherently bounded because it requires the number of desired digits to be specified up front as an input to the algorithm. In an article copyrighted in 2005, Jeremy Gibbons proposed a spigot algorithm based on Rabinowitz and Wagon's algorithm that doesn't have this limitation. It is unbounded and, as such, was an ideal candidate for this chapter's computationally intensive threads.

SUMMARY

A thread is a sequence of executable lines of code. Java programs can create and run more than one thread concurrently. Threads are organized into a tree structure composed of threads and thread groups. All Java programs are multi-threaded because the JVM and the Swing/AWT framework are themselves multi-threaded.

A thread can be created by extending `Thread` and overriding its `run()` method, or by passing a `Runnable` into one of the standard `Thread` constructors. A thread must be started by calling its `start()` method. Calling `Thread.sleep()` causes a thread to suspend execution and frees the CPU for other threads. A sleeping thread can be woken by calling `interrupt()`.

The JVM employs a preemptive algorithm for scheduling threads waiting to be run. Setting a thread's priority can affect how and when the JVM schedules it, which, in the face of other threads competing for CPU time, can affect the thread's performance. A thread should call `yield()` or `sleep()` or otherwise make provisions for sharing the CPU with competing threads. A thread that does not make provisions for sharing the CPU with competing threads (*by calling `yield()`, `sleep()` or some other mechanism*) is called a selfish thread. There is no guarantee that the underlying system will prevent selfish threads from taking complete control of the CPU. Because of variations in how systems handle the scheduling of threads, the correctness of a program's behavior should not rely on the particulars of any system.

A race condition is a condition where more than one thread accesses the same resource at the same time, resulting in errors and/or resource corruption. Synchronization can be used to prevent race conditions by ensuring that a block of code is executed by only one thread at a time. Synchronization is only effective when all threads attempting to execute a block of code are synchronized on a common object. Synchronization is applied to a block of code. Synchronized methods have a slightly different syntax but they too are simply blocks of code that have been synchronized.

In a producer-consumer relationship, a producer thread produces something that a consumer thread consumes and neither thread makes an effort to coordinate with the other. Coordination is handled instead by the object they both access. The `wait()` and `notify()` methods of `Object` can be used to negotiate the coordination between inter-dependent threads.

Deadlock is the situation where two or more threads remain permanently blocked because they are waiting for locks each other already holds. Deadlock should be avoided and cannot be easily detected. Unfortunately, the only method for avoiding it is to understand the threads you write and their potential interactions and to program in such a way that deadlock is never a possibility.

Skill-Building Exercises

- 1. Modifying a Thread:** Modify `ClockPanel1` to call `TreePrinterUtils.printThreads()` at the end of `main()` and also the tenth time `paintComponent()` is called. Notice what happened to the main thread?
- 2. Why Threads are Necessary:** The following code doesn't do what it was intended to do. Fix it. When the user clicks the "Start" button, the program should start incrementing the count variable. Whenever the user presses the "Update Display" button, the label should show the current value of count.

16.26 *chap16.exercises.BadCounter.java*

```

1      package chap16.exercises;
2
3      import java.awt.BorderLayout;
4      import java.awt.Container;
5      import java.awt.event.ActionEvent;
6      import java.awt.event.ActionListener;
7
8      import javax.swing.JButton;
9      import javax.swing.JFrame;
10     import javax.swing.JLabel;
11
12     public class BadCounter extends JFrame {
13         private long count = 0;
14         public BadCounter() {
15             Container contentPane = getContentPane();
16
17             JButton startButton = new JButton("Start");
18             startButton.addActionListener(new ActionListener() {
19                 public void actionPerformed(ActionEvent e) {
20                     while (count < Long.MAX_VALUE) {
21                         count++;
22                     }
23                 }
24             });
25
26             final JLabel countLabel = new JLabel("Press \"Start\" to start counting");
27
28             JButton updateButton = new JButton("Update Display");
29             updateButton.addActionListener(new ActionListener() {
30                 public void actionPerformed(ActionEvent e) {
31                     countLabel.setText(String.valueOf(count));
32                 }
33             });
34
35             contentPane.add(BorderLayout.NORTH, startButton);
36             contentPane.add(BorderLayout.CENTER, countLabel);
37             contentPane.add(BorderLayout.SOUTH, updateButton);
38
39             pack();
40             setDefaultCloseOperation(EXIT_ON_CLOSE);
41         }
42         public static void main(String[] arg) {
43             new BadCounter().show();
44         }
45     }

```

- 3. Shifting the Responsibility for Synchronization:** Modify `chap16.race.Breaker` to use an instance of `Synched-LongSetter` and verify that this approach prevents race conditions.
- 4. Choosing Your Locks:** Without changing `chap16.deadlock.Breaker` at all, rewrite `Caller` so that only one thread can call `Caller.call()` at the same time, only one thread can call `Caller.answer()` at the same time (this is already true of the first version of `Caller`) but without vulnerability to deadlock. Hint: use different locks. Test this by running `chap.16.deadlock.Breaker`.
- 5. (a) Producer-Consumer Practice:** Following the template below, implement `DigitHolder` in a new class named `BufferedDigitHolder`. It should maintain an array of digits. The consumer should be able to retrieve digits as long

as there are unconsumed digits and the producer should be able to store digits as long as there is space to put them without overwriting unretrieved digits. Modify PiPanelProdCons to use an instance of BufferedDigitHolder and test its behavior for various sizes between 1 and 1000. It should work correctly for any size greater than zero. Pay attention to System.out to make sure your BufferedDigitHolder is operating correctly.

16.27 chap16.exercises.BufferedDigitHolder.java

```

1      //insert package declarations and imports as desired
2      import chap16.prodcons.DigitHolder;
3
4      public class BufferedDigitHolder implements DigitHolder {
5          private final int[] digits;
6          //insert fields here as desired
7
8          public BufferedDigitHolder(int maxSize) {
9              //insert code here as desired
10             digits = new int[maxSize];
11         }
12         public synchronized void store(int digit) {
13             //insert code here as desired
14             System.out.println("store: size = " + size());
15         }
16         public synchronized int retrieve() {
17             int digit;
18             //insert code here as desired
19             System.out.println("retrieve: size = " + size());
20             return digit;
21         }
22         /**
23          * Returns the current number of stored, unretrieved digits
24          */
25         public synchronized int size() {
26             //insert code here as desired
27         }
28     }

```

5. (b) **Assimilating Your Knowledge:** Producing pi’s digits takes longer than consuming them, and it takes longer and longer as more and more digits have been produced. So, slow the consumer thread down to make it a little slower than the producer when the program starts. As the producer continues to produce digits, its speed will decrease until at some point it will be slower than the consumer. System.out should show the Buffered-DigitHolder’s size increase to its maximum size while the producer is quicker than the consumer. Then when the consumer’s speed overtakes the producers speed, you should see the BufferedDigitHolder’s size decrease until it is down to one digit.

SUGGESTED PROJECTS

1. **Combining Thread Skills with GUI Skills:** Write a simple, animated computer game like “Pong”. If you don’t know what “Pong” is, look it up on the internet.
2. **Investigating java.util.Timer.** Using java.util.Timer, write a reminder program that allows the user to specify any number of messages and a time to display each. The program should display each message at the appropriate time.

SELF-TEST QUESTIONS

1. Name and describe the methods declared by Thread that can cause a thread to stop/pause execution.
2. Name and describe the methods declared by Thread that might cause a thread to start/resume execution.

3. Name and describe the methods declared by Object that can cause a thread to stop/pause execution.
4. Name and describe the methods declared by Object that might cause a thread to start/resume execution.
5. What is a race condition?
6. How can race conditions be prevented?
7. What is deadlock?
8. How can deadlock be prevented?
9. What is a selfish thread?
10. What are the two ways to construct a Thread?
11. What are the purposes of a ThreadGroup?
12. Under what circumstances may a thread be preempted by another thread?
13. On what does a synchronized non-static method synchronize?
14. On what does a synchronized static method synchronize?

REFERENCES

Java 2 API Specification: [java.sun.com/j2se/1.4.2/docs/api]

The Java Tutorial: [java.sun.com/docs/books/tutorial]

Doug Lea, *Concurrent Programming in Java Second Edition*. The Java Series. Addison-Wesley. ISBN: 0-201-31009-0

Stanley Rabinowitz and Stan Wagon, *A Spigot Algorithm for the Digits of Pi*, American Mathematical Monthly, March 1995, 195-203.

Jeremy Gibbons, *An Unbounded Spigot Algorithm for the Digits of Pi*: [web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/spigot.pdf]

NOTES

CHAPTER 17



Sideways CHRISTMAS TREE

Collections

LEARNING OBJECTIVES

- *STATE THE PURPOSE AND USE OF THE JAVA COLLECTIONS FRAMEWORK*
- *LIST AND DESCRIBE THE COLLECTION INTERFACES AND CLASSES PROVIDED BY THE JAVA COLLECTIONS FRAMEWORK*
- *DESCRIBE THE GENERAL PERFORMANCE CHARACTERISTICS OF AN ARRAY*
- *DESCRIBE THE GENERAL PERFORMANCE CHARACTERISTICS OF A LINKED-LIST*
- *DESCRIBE THE GENERAL PERFORMANCE CHARACTERISTICS OF A HASHTABLE*
- *DESCRIBE THE GENERAL PERFORMANCE CHARACTERISTICS OF A RED-BLACK TREE*
- *DEMONSTRATE YOUR ABILITY TO UTILIZE COLLECTIONS IN YOUR JAVA PROGRAMS*
- *UTILIZE CASTING WHEN RETRIEVING AN OBJECT FROM A JAVA 1.4.X COLLECTION*
- *EXPLAIN WHY JAVA GENERIC COLLECTIONS REDUCE THE NEED TO CAST RETRIEVED OBJECTS*
- *DESCRIBE THE DIFFERENCES BETWEEN JAVA 1.4.X COLLECTIONS AND JAVA 5 COLLECTIONS*
- *DESCRIBE AND EMPLOY THE NEW SYNTAX REQUIRED TO DECLARE A JAVA 5 COLLECTION REFERENCE*
- *UTILIZE THE ENHANCED FOR LOOP TO ITERATE OVER A JAVA 5 COLLECTION*
- *DESCRIBE THE EFFECTS OF PRIMITIVE TYPE AUTOBOXING AND UNBOXING*

INTRODUCTION

When considering all functional assets of the Java Platform API, none possess the extraordinary potential to save you more time and hassle than the Java collections framework. Located in the `java.util` package, the interfaces, classes, algorithms, and infrastructure code belonging to the Java collections framework provide you with a unified way to manipulate collections of objects.

This chapter introduces you to the Java collections framework and shows you how to employ its interfaces and classes in your programs. Along the way you will learn the general characteristics of lists, sets, and maps, and the functional characteristics of linked lists, hashables, and red-black trees. Knowledge of these concepts will set the stage for understanding the inner workings of collections framework classes.

From the beginning, the Java Platform API has undergone continuous evolutionary improvement. Nowhere is this more evident than in the Java collections framework. Each subsequent release of the Java platform introduces improved collections capability.

Continuous evolutionary improvement is a good thing for programmers. Each new release of the Java API fixes bugs discovered in the previous release. Less bugs mean more stability. That's a good thing. On the other hand, improvements to the Java platform can present significant challenges to programmers. Each new release introduces, in some cases, sweeping new functionality. Such is the case between Java platform version 1.4.x and version 5.

The differences between the Java 5 collections framework and previous versions are significant. Java 5 introduced Java generics. It also introduced autoboxing and unboxing of primitive data types and an enhanced for loop for processing collections. These changes reduce and sometimes eliminate the need to utilize certain coding styles previously required to manipulate collections in earlier platform versions. For example, earlier versions of the Java collection classes stored `Object` references. When the time came to access references stored in a collection the `Object` reference retrieved had to be cast to the required interface. Such use of casting rendered code hard to read and proved difficult for novice programmers to master.

However, just because there is a new version of the API does not mean everybody switches to it right away. There is currently lots of new code being developed targeting Java platform version 1.4.x or earlier. Because of this I feel obligated to discuss the collections framework from two platform aspects: 1.4.x and 5.

I begin with a case study of a user-defined dynamic array class. The purpose of the case study is to provide a motivational context that demonstrates the need for a collections framework. I follow the case study with an overview of the collections framework, introducing you to its core interfaces and classes. I will then show you how to manipulate collections using Java platform 1.4.x programming techniques. You will find this treatment handy primarily because you may find yourself faced with maintaining legacy Java code. I will then discuss improvements made to the collections framework introduced with Java 5. This will include a discussion of static polymorphic behavior as it is realized by generics. I will also show you how to use the enhanced for loop and discuss autoboxing and unboxing of primitive types when being inserted into and retrieved from a collection.

CASE STUDY: BUILDING A DYNAMIC ARRAY

Imagine, for a moment, that you are working on a Java project and you're deep into the code. You're in the flow and you don't want to stop to read no stinkin' API documentation. The problem at hand dictates the need for an array with special powers — one that can automatically grow itself when one too many elements are inserted. To solve your problem you hastily crank out the code for a class named `DynamicArray` shown in example 17.1 along with a short test program shown in example 17.2. Figure 17-1 gives the results of running the test program.

17.1 DynamicArray.java

```
1     public class DynamicArray {
2         private Object[] _object_array = null;
3         private int _next_open_element = 0;
4         private int _growth_increment = 10;
5         private static final int INITIAL_SIZE = 25;
6
7         public DynamicArray(int size){
8             _object_array = new Object[size];
9         }
10    }
```

```

11     public DynamicArray(){
12         this(INITIAL_SIZE);
13     }
14
15     public void add(Object o){
16         if(_next_open_element < _object_array.length){
17             _object_array[_next_open_element++] = o;
18         }else{
19             growArray();
20             _object_array[_next_open_element++] = o;
21         }
22     } // end add() method;
23
24
25     private void growArray(){
26         Object[] temp_array = _object_array;
27         _object_array = new Object[_object_array.length + _growth_increment];
28         for(int i=0, j=0; i<temp_array.length; i++){
29             if(temp_array[i] != null){
30                 _object_array[j++] = temp_array[i];
31             }
32             _next_open_element = j;
33         }
34         temp_array = null;
35     } // end growArray() method
36
37
38     public Object get(int index){
39         if((index >= 0) && (index < _object_array.length)){
40             return _object_array[index];
41         }else return null;
42     }
43
44     public int size(){ return _object_array.length; }
45
46 } // end DynamicArray class definition

```

17.2 ArrayTestApp.java

```

1     public class ArrayTestApp {
2         public static void main(String[] args){
3             DynamicArray da = new DynamicArray();
4             System.out.println("Array size is: " + da.size());
5             da.add(new String("Ohhh if you loved Java like I love Java!!"));
6             System.out.println(da.get(0).toString());
7             for(int i = 1; i<26; i++){
8                 da.add(new Integer(i));
9             }
10            System.out.println("Array size is: " + da.size());
11            for(int i=0; i<da.size(); i++){
12                if(da.get(i) != null){
13                    System.out.print(da.get(i).toString() + ", ");
14                }
15            }
16            System.out.println();
17        } //end main() method
18    } // end ArrayTestApp class definition

```

```

Terminal - tcsh
[Rick-Millens-Computer:~/desktop/dynamic_array] swodog$ java ArrayTestApp
Array size is: 0
Ohhh if you loved Java like I love Java!!
Array size is: 26
Ohhh if you loved Java like I love Java!! 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
[Rick-Millens-Computer:~/desktop/dynamic_array] swodog$

```

Figure 17-1: Results of Testing DynamicArray

Referring to example 17.1 — the data structure used as the basis for the `DynamicArray` class is an ordinary array of Objects. Its initial size can be set via a constructor or, if the default constructor is called, the initial size is set to 25 elements. Its growth increment is 10 elements, meaning that when the time comes and the array must expand, it will expand by 10 elements. Besides its two constructors the `DynamicArray` class has four additional methods: `add()`, `growArray()`, `get()`, and `size()`.

The `add()` method inserts an `Object` reference into the next available array element which is pointed to by the `_next_open_element` variable. If there's no room left in the array the `growArray()` method is called to grow the array.

The `growArray()` method must create a temporary array of Objects and copy each element to the temporary array. It must then create a new, larger Object array and copy the elements to it from the temporary array.

The `get()` method allows you to access each element of the array. If the index argument falls out of bounds the method returns null. The `size()` method simply returns the number of elements (*references*) contained in the array, which is the value of the `_next_open_element` variable.

Referring to example 17.2 — the `ArrayTestApp` class is a short program that tests the functionality of the `DynamicArray` class. On line 3 an instance of `DynamicArray` is created using the default constructor. This will result in an initial array length of 25 elements. Initially, its size will be zero because no references have yet been inserted. On line 5 a `String` object is added to the array and then printed to the console on line 6. The `for` statement on line 7 will insert enough `Integer` objects to test the array's growth capabilities. The `for` statement on line 11 prints all the non-null elements to the console.

EVALUATING DYNAMICARRAY

The `DynamicArray` class works well enough for your immediate needs but it suffers several shortcomings that will cause serious problems should you try to employ the `DynamicArray` class in more demanding situations. For example, although you can access each element of the array you cannot remove elements. You could add a method called `remove()` but what happens when the number of remaining elements falls below a certain threshold? You might want to shrink the array as well.

Another point to consider is how to insert references into specific element locations. When this happens you must make room for the reference at the specified array index location and shift the remaining elements to the right. If you plan to frequently insert elements into your custom-built `DynamicArray` class you will have a performance issue on your hands you did not foresee.

At this point you would be well served to take a break from coding and dive into the API documentation to study up on the Java Collections Framework. There you will find that all this work, and more, is already done for you!

THE ARRAYLIST CLASS TO THE RESCUE

Let's re-write the `ArrayTestApp` program with the help of the `List` interface and the `ArrayList` class, both of which belong to the Java collections framework. Example 17.3 gives the code. Figure 17-2 shows the results of running this program.

17.3 *ArrayTestApp.java (Mod 1)*

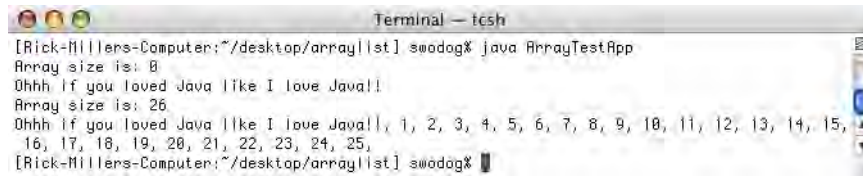
```

1      import java.util.*;
2
3      public class ArrayTestApp {
4          public static void main(String[] args){
5              List da = new ArrayList();
6              System.out.println("Array size is: " + da.size());
7              da.add(new String("Ohhh if you loved Java like I love Java!!"));
8              System.out.println(da.get(0).toString());
9              for(int i = 1; i<26; i++){
10                 da.add(new Integer(i));
11             }
12             System.out.println("Array size is: " + da.size());
13             for(int i=0; i<da.size(); i++){
14                 if(da.get(i) != null){
15                     System.out.print(da.get(i).toString() + ", ");
16                 }
17             }
18             System.out.println();
19         } //end main() method
20     } // end ArrayTestApp class definition

```

Referring to example 17.3 — only three changes were made to the original `ArrayTestApp` program: 1) the import statement now appears on line 1 to provide shortcut naming to the interfaces and classes of the `java.util` package, 2) the `da` reference declared on line 5 has been changed from a `DynamicArray` type to a `List` type, and 3) also on line 5, an instance of `ArrayList` is created instead of a `DynamicArray`.

If you compare figures 17-1 and 17-2 you will see that the output produced using an `ArrayList` is exactly the same as that produced using the `DynamicArray`. However, the `ArrayList` class provides much more ready-made functionality.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/arraylist] swadog% java ArrayTestApp
Array size is: 0
Ohhh if you loved Java like I love Java!!
Array size is: 26
Ohhh if you loved Java like I love Java!! 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
[Rick-Millers-Computer:~/desktop/arraylist] swadog%

```

Figure 17-2: Results of Running Example 17.3

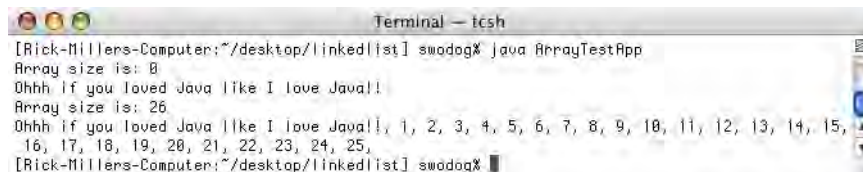
The use of a Java collections framework interface in the revised `ArrayTestApp` program also provides another point of flexibility. Because the `da` reference declared on line 5 is of type `List`, any collections framework class that implements the `List` interface can be used to actually perform the work. For example, suppose you decide that you will need to make lots of insertions and deletions of elements into the collection. Since `ArrayList` utilizes an array as its underlying data structure, just like `DynamicArray`, it is not best suited to perform under these conditions. However, by making one more small change to the `ArrayTestApp` program you can swap out the `ArrayList` with a `LinkedList` and thus solve the problem. Example 17.4 gives the revised code for the `ArrayTestApp` program. Figure 17-3 shows the results of running this program.

17.4 `ArrayTestApp.java (Mod 2)`

```

1      import java.util.*;
2
3      public class ArrayTestApp {
4          public static void main(String[] args){
5              List da = new LinkedList();
6              System.out.println("Array size is: " + da.size());
7              da.add(new String("Ohhh if you loved Java like I love Java!!"));
8              System.out.println(da.get(0).toString());
9              for(int i = 1; i<26; i++){
10                 da.add(new Integer(i));
11             }
12             System.out.println("Array size is: " + da.size());
13             for(int i=0; i<da.size(); i++){
14                 if(da.get(i) != null){
15                     System.out.print(da.get(i).toString() + ", ");
16                 }
17             }
18             System.out.println();
19         } //end main() method
20     } // end ArrayTestApp class definition

```



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/linkedlist] swadog% java ArrayTestApp
Array size is: 0
Ohhh if you loved Java like I love Java!!
Array size is: 26
Ohhh if you loved Java like I love Java!! 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
[Rick-Millers-Computer:~/desktop/linkedlist] swadog%

```

Figure 17-3: Results of Running Example 17.4

Referring to example 17.4 — the only change between this and the previous version of `ArrayTestApp` is the substitution on line 5 of `LinkedList` for `ArrayList`. As you can see from looking at figure 17-3 the results produced are exactly the same. The implementation differences, however, between the `LinkedList` and `ArrayList` classes are significant, but those differences are hidden from you since, to paraphrase an old saying, a `List`, by any other name, is still a `List`.

THE POWER OF COLLECTIONS – POLYMORPHIC CODE

The last two versions of `ArrayTestApp` provide a glimpse of the potential power gained by proper use of collections framework classes. By targeting collections framework interfaces you can write code that works with any collection class that implements that interface. The term *polymorphic code*, as it applies to object-oriented programming, means writing code that relies on interfaces vs. implementations. Thus, objects with the same interface but different implementations can be used in your code and your code will continue to work as designed. (*i.e.*, *It will not break!*)

JAVA COLLECTIONS FRAMEWORK OVERVIEW

The Java collections framework provides interfaces, implementation classes, algorithms, and infrastructure code designed to make the manipulation of collections of objects safe, reliable, and easy. This section describes the purpose and organization of the Java collections framework. It also discusses the benefits you can expect to derive from utilizing the collections framework in your Java programs.

THE PURPOSE OF THE COLLECTIONS FRAMEWORK

Suppose for a moment the Java collections framework did not exist. Every Java programmer who needed to manage a collection of objects would have to create custom-designed classes like `DynamicArray` to satisfy their needs. These custom-designed classes would have different interface methods, questionable performance characteristics, and dubious track records from lack of rigorous and proper testing. Somewhere along the line one or more enterprising developers would offer up their custom-designed classes for use by other programmers. Some of these APIs might be open source and others might be offered as commercial products. Some would perform better than others and all might have unique features not found in the other APIs. Java programmers would be faced with learning one or more collections APIs with an eye towards getting best-of-breed performance. There would be no guarantees regarding the interoperability between different APIs. The life of the Java programmer would be a mess indeed!

The Java collections framework directly addresses these issues by providing a comprehensive API for collections management. Java developers learn one API and their collection manipulation code is guaranteed to work across deployment platforms. Custom classes can be built by implementing the required collection interface and following a few simple rules. Java developers can write more reliable code because the collections framework has been subjected to rigorous testing by the entire Java developer community. Life is good!

FRAMEWORK ORGANIZATION

The Java collections framework is organized into four primary areas:

- 1) a set of *core collection interfaces*
- 2) a set of *general purpose implementation classes*
- 3) ready-made *algorithms* for collection manipulation
- 4) *infrastructure* code to facilitate collection manipulation

To gain full benefit from the collections framework you need to know the following things:

- 1) the basic characteristics of each core interface and the methods supported by each
- 2) the general purpose implementation classes and their fundamental differences
- 3) how to manipulate collections using the supplied algorithms
- 4) how objects are ordered in collections
- 5) how to manipulate collections using iterators

CORE COLLECTIONS INTERFACES

Figure 17-4 shows the core collections interface hierarchy for the Java 1.4.2 platform. As you can see from the diagram there are six core interfaces grouped into two distinct inheritance hierarchies: one with the `Collection` interface as the root, and the other with the `Map` interface as the root. Another point illustrated in figure 17-4 is that a `Map` is not a `Collection` per se, however, the `Map` interface does declare methods that return different views of a `Map` as a `Collection` object. A `Map` whose elements have been converted to a `Collection` can then be manipulated accordingly.

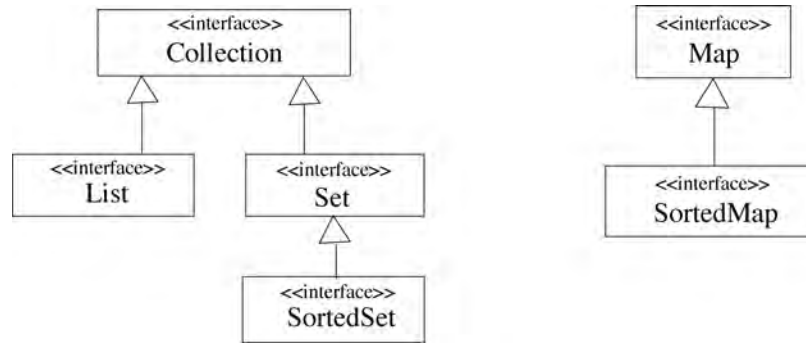


Figure 17-4: Java 1.4.2 Collections Framework Core Interface Hierarchy

Table 17-1 lists the general characteristics of each Java collections framework core interface. An understanding of each interface's characteristics will help you select the right type of collection to suit your programming needs.

Interface	Characteristic
Collection	The Collection interface is the base interface of all collections. Its purpose is to provide a unified way to manipulate collection objects. Some of the methods declared by the Collection interface are optional. An optional method may, or may not, be implemented by a general purpose collection implementation class. As with all collection classes, you should consult the API documentation to learn which methods are supported. A Collection represents a group of objects referred to as its <i>elements</i> . The elements of a collection can be accessed and manipulated via an <i>Iterator</i> .
List	The List interface represents a collection whose elements are arranged sequentially. Elements can be inserted into the list at any location. Duplicate elements are usually allowed. List elements can be accessed and manipulated via a <i>ListIterator</i> . A <i>ListIterator</i> is different from an ordinary <i>Iterator</i> in that it provides the means to traverse the list forwards and backwards.
Set	The Set interface represents a collection that contains no duplicate elements.
SortedSet	The SortedSet interface represents a Set whose <i>Iterator</i> will traverse its elements according to their natural ordering. Insertions into a sorted set may take longer but retrievals will occur within predictable time.
Map	The Map interface represents an object that stores and retrieves an element (<i>value</i>) whose location within the map is determined by a <i>key</i> . A Map cannot contain duplicate values and a key can map to only one value. Note that a Map is not a Collection although it does provide methods that return a Collection object that contains the map's elements. The Collection object returned in this fashion can be manipulated accordingly.
SortedMap	The SortedMap interface represents a Map whose elements are stored according to their natural ordering. Insertions into a SortedMap may take longer but retrievals will occur within predictable time.

Table 17-1: Core Collection Interface Characteristics

GENERAL PURPOSE IMPLEMENTATION CLASSES

The Java collections framework provides a set of general purpose implementation classes ready for immediate use in your programs. These classes implement the core collection interfaces, so you have, at your disposal, lists, sets, sorted sets, maps and sorted maps.

There are generally two or more flavors of each implementation class. Each flavor is based upon a fundamental underlying data structure such as an array, linked list, or a tree. (*Here I use the words flavor and subtype synonymously.*) To better understand how to choose the right collection class to suit your needs you must understand the general performance characteristics of each of these underlying data structures.

ARRAY PERFORMANCE CHARACTERISTICS

As you know already from reading chapter 8, an array is a contiguous collection of homogeneous elements. You can have arrays of primitive types or arrays of references to objects. The general performance issues to be aware of regarding arrays concern inserting new elements into the array at some position prior to the last element, accessing elements, and searching for particular values within the array.

When a new element is inserted into an array at a position other than the end, room must be made at that index location for the insertion to take place by shifting the remaining references one element to the right. This series of events is depicted in figures 17-5 through 17-7.

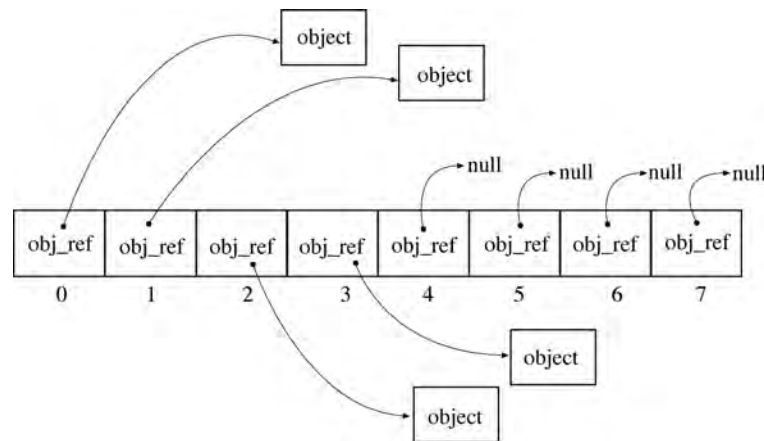


Figure 17-5: Array of Object References Before Insertion

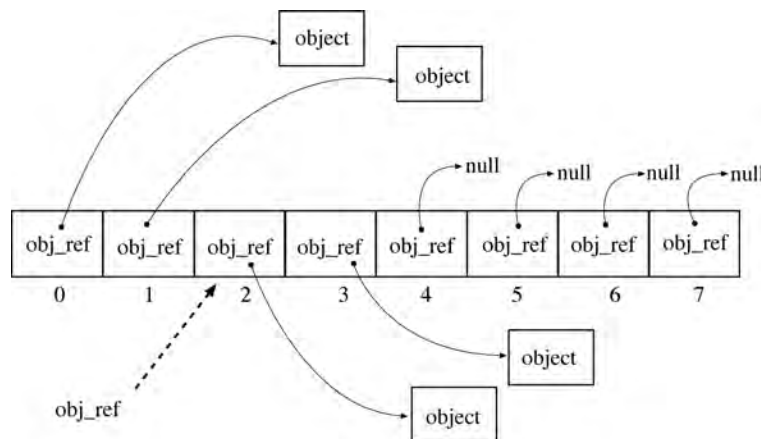


Figure 17-6: New Reference to be Inserted at Array Element 3 (index 2)

Referring to figures 17-5 through 17-7 — an array of object references contains references that may point to an object or to null. In this example array elements 1 through 4 (*index values 0 through 3*) point to objects while the remaining array elements point to null.

A reference insertion is really just an assignment of the value of the reference being inserted to the reference residing at the target array element. To accommodate the insertion, the values contained in references located to the right of the target element must be reassigned one element to the right. (*i.e., They must be shifted to the right.*) It is this shifting action which can cause a performance hit when inserting elements into an array-based collection. If the insertion triggers the array growth mechanism, then you'll receive a double performance hit. The insertion performance penalty, measured in time, grows with the length of the array. Element retrieval, on the other hand, takes place fairly quickly because of the way array element addresses are computed.

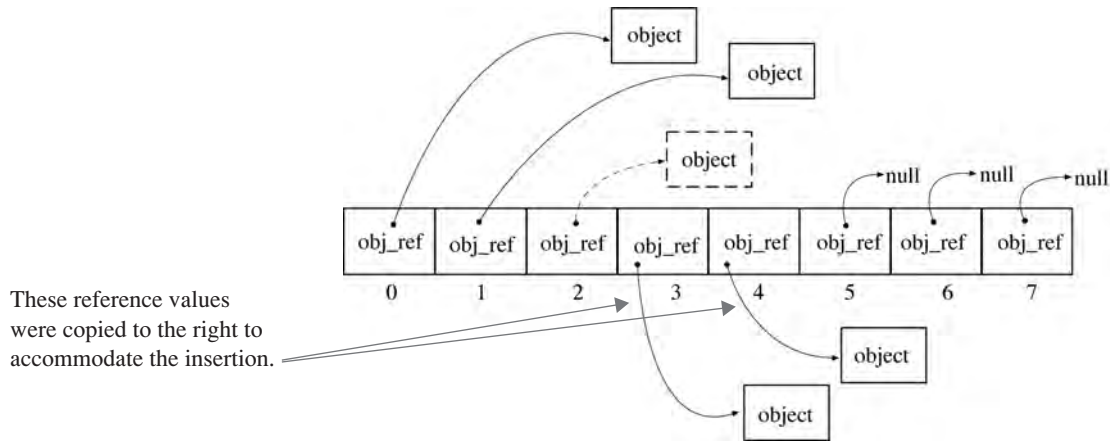


Figure 17-7: Array After New Reference Insertion

LINKED LIST PERFORMANCE CHARACTERISTICS

A linked list is a data structure whose elements stand alone in memory. (And may indeed be located anywhere in the heap!) Each element is linked to another by a reference. Unlike the elements of an array, which are ordinary references, each linked list node is a complex data structure that contains a reference to the previous node in the list, the next node in the list, and a reference to an object payload as figure 17-8 illustrates.

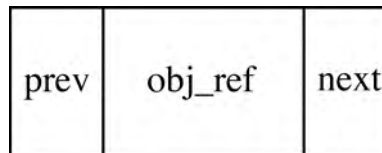


Figure 17-8: Linked List Node Organization

So, whereas an array’s elements are always located one right after the other in memory, and thus their memory addresses can be quickly calculated, the linked list’s elements can be, and usually are, scattered in memory hither and yonder. The nice thing about linked lists is that element insertions can take place fairly quickly because no element shifting is required. Figures 17-9 through 17-11 show the sequence of events for a circular linked list node insertion.

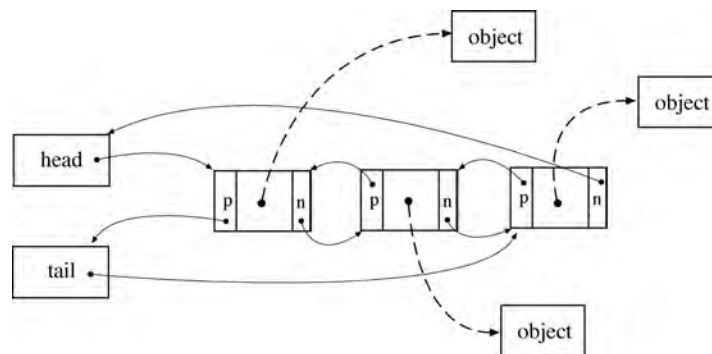


Figure 17-9: Linked List Before New Element Insertion

Referring to figures 17-9 through 17-11 — a linked list can contain one or more non-contiguous nodes. A node insertion requires the rewiring of the references involved. This includes setting the previous and next references on the new node in addition to resetting the affected references of its adjacent list nodes. If this looks complicated guess what? It is! And if you take a data structures class you’ll get the chance to create a linked list from scratch!

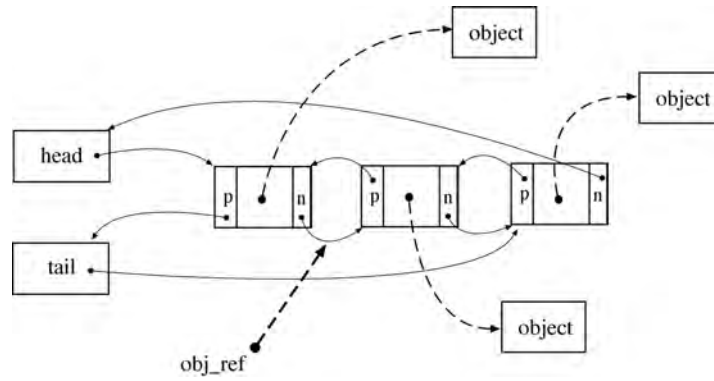


Figure 17-10: New Reference Being Inserted Into Second Element Position

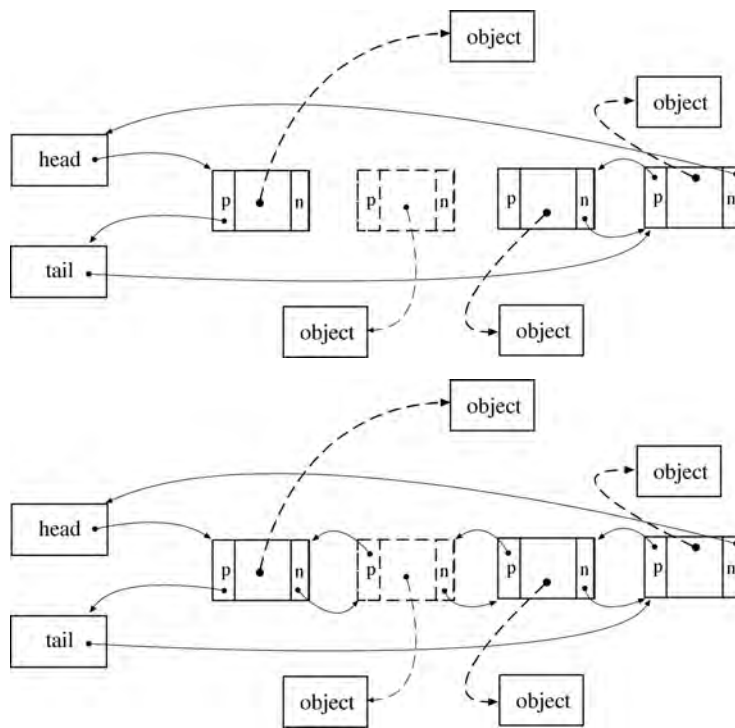


Figure 17-11: References of Previous, New, and Next List Elements Must Be Manipulated

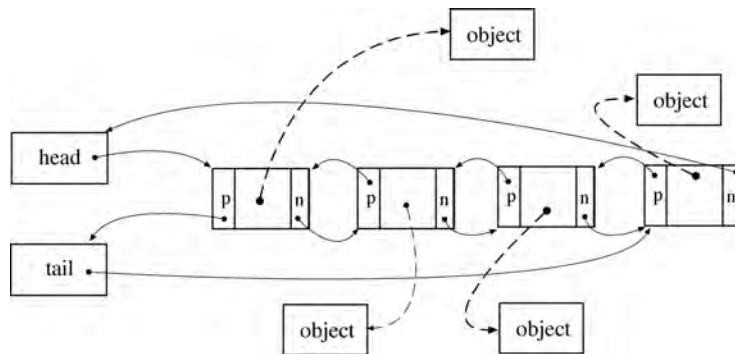


Figure 17-12: Linked List Insertion Complete

HashTable Performance Characteristics

A hashtable is an array whose elements can point to a series of nodes. Structurally, as you'll see, a hashtable is a cross between an array and a one-way linked list. In an ordinary array elements are inserted by index value. If there are potentially many elements to insert the array space required to hold all the elements would be correspondingly large as well. This may result in wasted memory space. The hashtable addresses this problem by reducing the size of the array used to point to its elements and assigning each element to an array location based on a hash function as figure 17-13 illustrates.

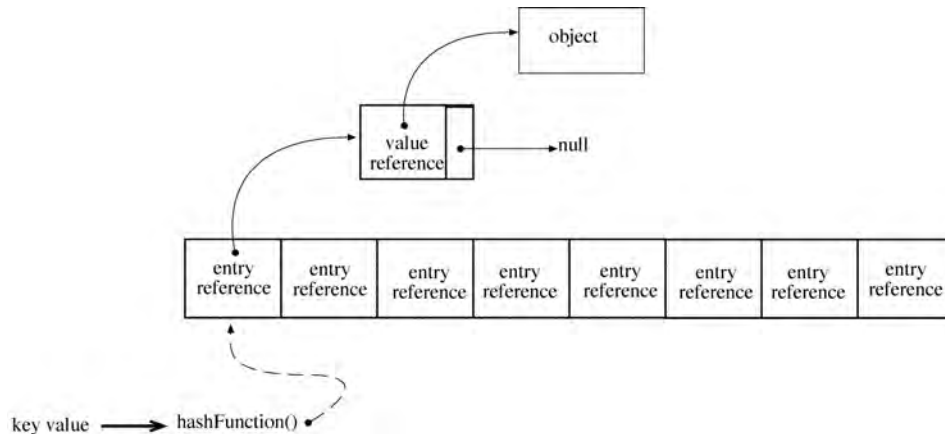


Figure 17-13: A Hash Function Transforms a Key Value into an Array Index

Referring to figure 17-13 — the purpose of the hash function is to transform the key value into a unique array index value. However, sometimes two unique key values will translate to the same index value. When this happens a *collision* is said to have occurred. This problem is resolved by chaining together nodes that share the same hashtable index as is shown in figure 17-14.

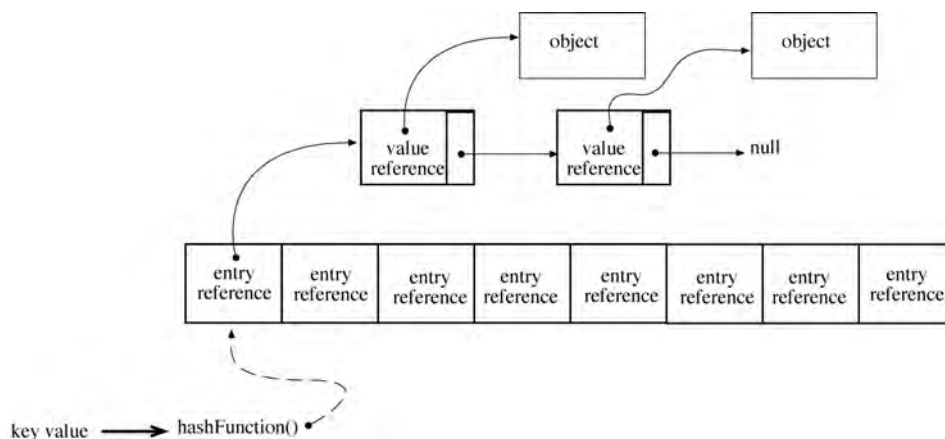


Figure 17-14: Hash Table Collisions are Resolved by Linking Nodes Together

The benefits of a hashtable include lower initial memory overhead and relatively fast element insertions. On the other hand, if too many insertion collisions occur, the linked elements must be traversed to insert new elements or to retrieve existing elements. List traversal extracts a performance penalty.

Red-Black Tree Performance Characteristics

Collections that have the word “tree” in their name implement the SortedSet or SortedMap interface and automatically sort their elements upon insertion. These automatically sorting collections are implemented using an underlying data structure known as a red-black tree. A red-black tree is a type of binary search tree with a self-balancing

characteristic. Tree nodes have an additional data element, color, that can be set to either red or black. The data elements of a red-black tree node are shown in figure 17-15.

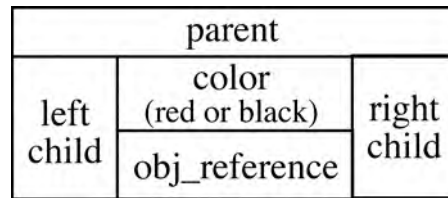


Figure 17-15: Red-Black Tree Node Data Elements

Insertions into a red-black tree are followed by a self-balancing operation that ensures all leaf nodes are the same number of black nodes away from the root node. Figure 17-16 shows the state of a red-black tree after inserting the integer values 1 through 9 in the following insertion order: 9, 3, 5, 6, 7, 2, 8, 4, 1. (*Red nodes shown lightly shaded.*)

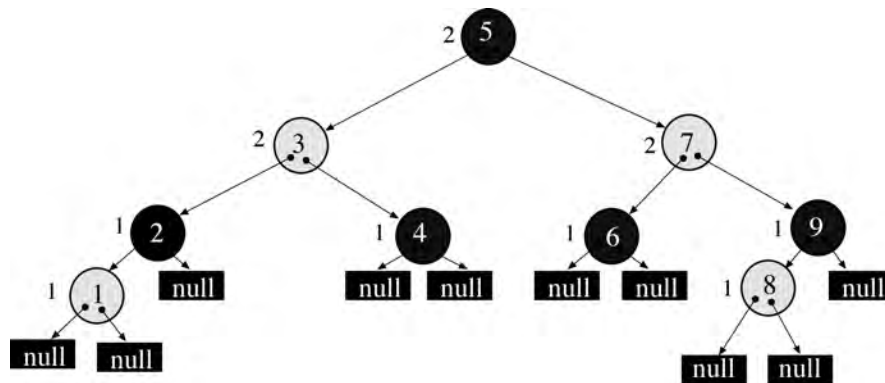


Figure 17-16: Red-Black Tree After Inserting Integer Values 9, 3, 5, 6, 7, 8, 4, 1

Referring to figure 17-16 — the numbers appearing to the left of each node represent the height of the tree in black nodes. The primary benefit associated with a red-black tree is the generally overall good node search performance that can be obtained regardless of the number of nodes the tree contains. However, because the tree reorders itself with each insertion, an insertion into a tree that contains lots of nodes will incur a performance penalty.

Think of it in terms of a clean room versus a messy room. You can store things really fast in a messy room because you just throw your stuff anywhere. Finding things in a messy room takes some time. You may have to look high and low before finding what you desire. Storing things in a clean room, conversely, takes a little while longer, but when you need something you can get to it in a hurry!

MAPPING AN IMPLEMENTATION CLASS TO ITS UNDERLYING DATA STRUCTURE

Now that you have a better idea of the performance characteristics associated with the fundamental data structures used in the Java collections framework general purpose implementation classes you need to know how to correlate the implementation class to its underlying data structure. That way you will know what performance characteristics to expect based on the class you choose.

The general purpose implementation class names are formulated by combining the name of the underlying data structure (*i.e.*, *Array*, *Linked*, *Hash*, or *Tree*) followed by the name of the interface the class implements. (*i.e.*, *List*, *Set*, or *Map*). For example, a *HashSet* is a *Set* whose underlying data structure is a hashtable; a *TreeSet* is a *Set* whose underlying data structure is implemented as a red-black tree.

Algorithms

The Java collections framework provides a ready-made set of algorithms you can use to manipulate collections. These algorithms are implemented as static methods in the *Collections* class. (*That's Collections with an 's'.*) The *Collections* class provides methods to sort and search collections as well as find the maximum and minimum element

values. The full range of Collections class functionality is too large to treat fully here, however, I recommend you explore the methods of the Collections class and get a feel for what they can do for you.

The *Arrays* class provides static methods to sort, search, and manipulate arrays. (*I introduced you to the functionality of the Arrays class formally in chapter 8.*) The Collection interface declares a method named `toArray()` that returns a collection's elements as an array of Objects or as an array of a specified type. Once in the form of an array, the elements can be manipulated using the static methods provided by the Arrays class.

INFRASTRUCTURE

The Java collections framework provides several classes and interfaces whose purpose is to help you manipulate collection elements (*Iterator* and *ListIterator*), create collection-aware classes (*Comparable* and *Comparator*), and catch collection-specific exceptions (*UnsupportedOperationException* and *ConcurrentModificationException*). Because these classes and interfaces are used across the entire collections framework they are referred to as infrastructure.

Quick Review

The purpose of the Java collections framework is to provide Java programmers with a unified, comprehensive way to manipulate collections of objects in their programs across deployment platforms. The Java collections framework is organized into four primary areas: 1) core collection interfaces, 2) general purpose implementation classes, 3) algorithms, and 4) infrastructure.

The core collection interfaces consist of Collection, List, Set, SortedSet, Map, and SortedMap. Map and SortedMap are not collections but have methods that return their elements in the form of a Collection object. The general purpose implementation classes employ arrays, linked lists, hashables, and red-black trees as their underlying data structures.

Array elements can be quickly retrieved but insertions into the array at element positions other than the end can extract a performance penalty because element values to the right of the insertion must be copied to the right.

Linked lists overcome the problems associated with array element insertion, however, the linked list must be traversed to find a desired element. List traversal extracts a performance penalty. Hashables are a cross between an array and a linked list.

Hashtable element storage locations are assigned based on the results of a hash function. Hashtable collisions are resolved by linking together elements that share the same hashtable element location. Hashables can save storage space but a poor hash function may result in frequent hashtable collisions which will result in a list traversal performance penalty.

Red-black trees store their elements in sorted order upon insertion. Element insertion times grow with tree size. However, tree element access time is good overall regardless of tree size.

The Collections class contains static methods that implement collection manipulation algorithms. The Arrays class contains static methods that can be used to manipulate arrays. A collection's elements can be converted into an array and in this form can be manipulated by the static methods of the Arrays class.

Infrastructure classes and interfaces are employed across the entire collections framework.

JAVA 1.4.2 STYLE COLLECTIONS

Java collections frameworks leading up to and including Java 1.4.2 allowed you to manage collections of objects without specifying what type of objects actually were contained in the collection. This led to the need to cast retrieved objects to the desired type if a method you wanted to call on the object was not one that overrode one of the methods defined by the Object class. This section presents several sample programs that illustrate typical collections framework programming style in Java platform versions prior to Java 5.


```

5     private String middle_name = null;
6     private String last_name = null;
7     private Calendar birthday = null;
8     private String gender = null;
9
10    public static final String MALE = "Male";
11    public static final String FEMALE = "Female";
12
13    public Person(String f_name, String m_name, String l_name, int dob_year,
14                  int dob_month, int dob_day, String gender){
15        first_name = f_name;
16        middle_name = m_name;
17        last_name = l_name;
18        this.gender = gender;
19        birthday = Calendar.getInstance();
20        birthday.set(dob_year, dob_month, dob_day);
21    }
22
23    public int getAge(){
24        Calendar today = Calendar.getInstance();
25        int now = today.get(Calendar.YEAR);
26        int then = birthday.get(Calendar.YEAR);
27        return (now - then);
28    }
29
30    public String getFullName(){ return (first_name + " " + middle_name + " " + last_name); }
31
32    public String getFirstName(){ return first_name; }
33    public void setFirstName(String f_name) { first_name = f_name; }
34
35    public String getMiddleName(){ return middle_name; }
36    public void setMiddleName(String m_name){ middle_name = m_name; }
37
38    public String getLastName(){ return last_name; }
39    public void setLastName(String l_name){ last_name = l_name; }
40
41    public String getNameAndAge(){ return (getFullName() + " " + getAge()); }
42
43    public String getGender(){ return gender; }
44
45    public void setBirthday(int year, int month, int day){ birthday.set(year, month, day); }
46
47    public String toString(){
48        return getFullName() + " " + getGender() + " " + getAge();
49    }
50
51 } //end Person class

```

17.7 PeopleManager.java

```

1     import java.util.*;
2
3     public class PeopleManager {
4         List people_list = null;
5
6         public PeopleManager(){
7             people_list = new LinkedList();
8         }
9
10        public void addPerson(String f_name, String m_name, String l_name, int dob_year,
11                              int dob_month, int dob_day, String gender){
12            people_list.add(new Person(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender));
13        }
14
15        public void deletePersonAtIndex(int index){
16            people_list.remove(index);
17        }
18
19        public void insertPersonAtIndex(int index, String f_name, String m_name, String l_name,
20                                       int dob_year, int dob_month, int dob_day, String gender){
21            people_list.add(index, new Person(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender));
22        }
23
24        public void listPeople(){
25            for(Iterator i = people_list.iterator(); i.hasNext();){
26                System.out.println(i.next().toString());
27            }
28        }
29    } // end PeopleManager class

```

17.8 PeopleManagerApplication.java

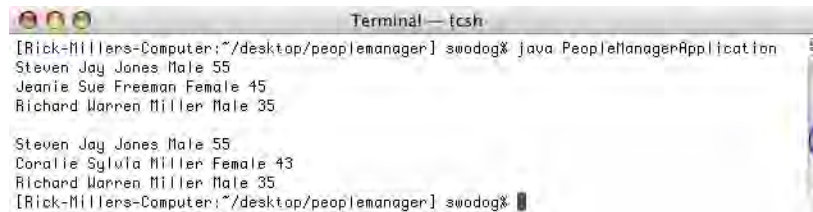
```

1      public class PeopleManagerApplication {
2          public static void main(String[] args){
3              PeopleManager pm = new PeopleManager();
4
5              pm.addPerson("Steven", "Jay","Jones", 1950, 8, 30, Person.MALE);
6              pm.addPerson("Jeanie", "Sue", "Freeman", 1960, 10, 10, Person.FEMALE);
7              pm.addPerson("Richard", "Warren", "Miller", 1970, 2, 29, Person.MALE);
8
9              pm.listPeople();
10
11             pm.deletePersonAtIndex(1);
12             pm.insertPersonAtIndex(1, "Coralie", "Sylvia", "Miller", 1962, 8, 3, Person.FEMALE);
13
14             System.out.println();
15             pm.listPeople();
16
17             } // end main
18         } // end PeopleManagerApplication class

```

Referring to example 17.7 — in the `PeopleManager` class the array has been replaced with a `LinkedList`. Also, since a `LinkedList` has no initial capacity setting, the constructor used to set the size of the array was removed because it was unnecessary. If you compare this version of `PeopleManager` to the original you will also find that I have removed the assertions as well. Because of the `toString()` method being added to the `Person` class there is no need to cast the retrieved objects in the body of the `listPeople()` method.

Referring to example 17.8 — the `PeopleManagerApplication` class adds three `Person` objects to the `PeopleManager` class. It then calls the `listPeople()` method on line 9. It then deletes the `Person` object at element 1, adds a new `Person` in that element location, and then calls `listPeople()` one last time. The results of running this program are shown in figure 17-18.



```

Terminal — [csh]
[Rick-Millers-Computer:~/desktop/peoplemanager] swoadog% java PeopleManagerApplication
Steven Jay Jones Male 55
Jeanie Sue Freeman Female 45
Richard Warren Miller Male 35

Steven Jay Jones Male 55
Coralie Sylvia Miller Female 43
Richard Warren Miller Male 35
[Rick-Millers-Computer:~/desktop/peoplemanager] swoadog%

```

Figure 17-18: Results of Running Example 17.8

CASTING RETRIEVED OBJECTS

So far, in each of the previous examples casting retrieved objects to their proper type was not required because the only methods called on the retrieved objects were those defined by the `Object` class (*i.e.* `toString()`) and overridden in the derived classes `Integer` and `Person`. In the next example the `Person` class is again used to illustrate how a retrieved object must be cast to the required type to call a method unique to that type. In this example I will store a few `Person` objects directly into a `LinkedList`.

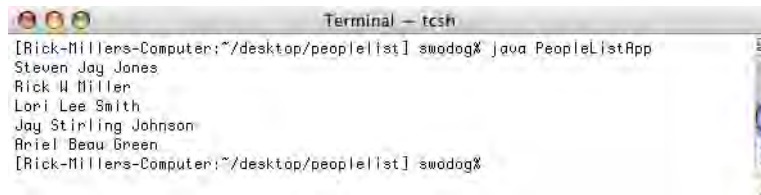
17.9 PeopleListApp.java

```

1      import java.util.*;
2
3      public class PeopleListApp {
4          public static void main(String[] args){
5              List list = new LinkedList();
6
7              list.add(new Person("Steven", "Jay","Jones", 1950, 8, 30, Person.MALE));
8              list.add(new Person("Rick", "W","Miller", 1963, 9, 22, Person.MALE));
9              list.add(new Person("Lori", "Lee","Smith", 1986, 1, 19, Person.MALE));
10             list.add(new Person("Jay", "Stirling","Johnson", 1947, 2, 02, Person.MALE));
11             list.add(new Person("Ariel", "Beau","Green", 1990, 3, 07, Person.MALE));
12
13             for(Iterator i = list.iterator(); i.hasNext();){
14                 System.out.println(((Person)i.next()).getFullName());
15             }
16             } // end main()
17         } // end PeopleListApp class definition

```

Referring to example 17.9 — a `LinkedList` is created on line 5 and five `Person` objects are added to the collection. The for loop beginning on line 13 uses an `Iterator` to step through the collection. Since the `Person.getFullName()` method is unique to the `Person` class, the objects retrieved from the collection must be cast to the `Person` type before the `getFullName()` method can be successfully called. Figure 17-19 shows the results of running this program.



```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/people|list] swadog% java PeopleListApp
Steven Jay Jones
Rick W Miller
Lori Lee Smith
Jay Stirling Johnson
Ariel Beau Green
[Rick-Millers-Computer:~/desktop/people|list] swadog%

```

Figure 17-19: Results of Running Example 17.9

CREATING NEW DATA STRUCTURES FROM EXISTING COLLECTIONS

Although the Java collections framework contains a wide variety of general purpose collection classes you will sometimes encounter a need for a data structure that does not exist in the collection. One example of this is the need for a queue. Since the Java 1.4.2 collections framework does not provide a `Queue` class one can be created based upon the functionality provided by the existing `LinkedList` class. The following example presents a class named `ActiveQueue` which is based on a `LinkedList`. `ActiveQueue` is a first-in-first-out (FIFO) queue that notifies interested listeners of queue insertion events. These interested listeners can then retrieve the longest waiting element. There are three parts to this example: `ActiveQueue`, `QueueListenerInterface`, and `QTesterApp`. These three classes are presented in examples 17.10 through 17.12.

17.10 *ActiveQueue.java*

```

1      /*****
2      ActiveQueue class implements a thread-safe FIFO queue
3      utilizing the java.util.LinkedList class.
4      *****/
5
6      import java.util.*;
7
8
9      /*****
10     ActiveQueue classs
11     @author Rick Miller
12     *****/
13
14     public class ActiveQueue {
15
16         private LinkedList _list = null;
17         private Vector _listeners = null;
18
19         /*****
20         Default constructor
21         *****/
22         public ActiveQueue(){
23             _list = new LinkedList();
24             _listeners = new Vector();
25         }
26
27
28         /*****
29         get() - returns the object that's been waiting the longest.
30         *****/
31         public synchronized Object get(){
32             return _list.getLast();
33         }
34
35         /*****
36         put() - inserts an object into the queue.
37         *****/
38         public synchronized void put(Object o){
39             _list.addFirst(o);
40             this.fireQueueInsertionEvent();
41         }
42
43         /*****

```



```

44     removeLast() - private method that removes the last element.
45     *****/
46
47     private void removeLast(){
48         _list.removeLast();
49     }
50
51     /*****
52     addQueueListener(QueueListenerInterface listener) adds the
53     argument QueueListener object to the _listeners Vector.
54     *****/
55     public void addQueueListener(QueueListenerInterface listener){
56         _listeners.add(listener);
57     }
58
59     /*****
60     fireQueueIntertionEvent() is a protected method called in
61     the body of the put() method to notify interested object
62     that something has been inserted into the Queue.
63     *****/
64     protected synchronized void fireQueueInsertionEvent(){
65         Vector v = (Vector) _listeners.clone();
66         for(Iterator i = v.iterator(); i.hasNext();){
67             ((QueueListenerInterface)i.next()).queueInsertionPerformed();
68         }
69         this.removeLast();
70     }
71 } // end ActiveQueue class definition

```

17.11 QueueListenerInterface.java

```

1     /*****
2     QueueListenerInterface - implemented by objects interested in
3     getting notificaiton when objects have been inserted into an
4     ActiveQueue object.
5     *****/
6     interface QueueListenerInterface {
7         void queueInsertionPerformed();
8     }

```

17.12 QTesterApp.java

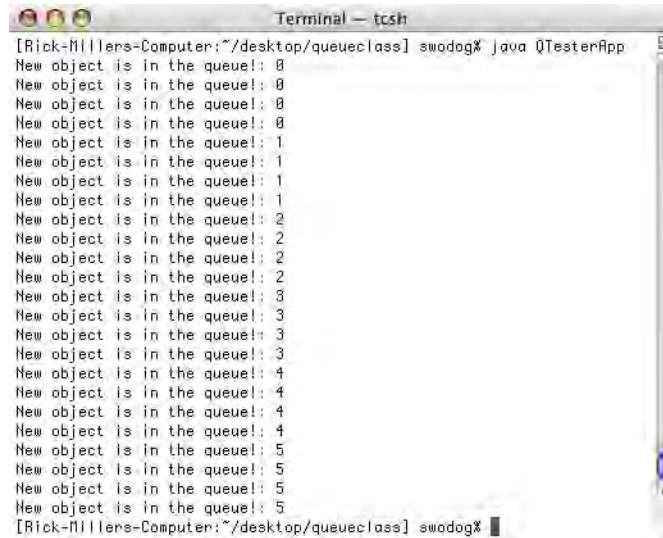
```

1     public class QTesterApp extends Thread implements QueueListenerInterface {
2
3         private static ActiveQueue _q = null;
4
5         static {
6             _q = new ActiveQueue();
7         }
8
9         public QTesterApp(){
10            _q.addQueueListener(this);
11        }
12
13        public void queueInsertionPerformed(){
14            System.out.print("New object is in the queue!: ");
15            System.out.println(_q.get().toString());
16        }
17
18        private void insertStuff(){
19            for(int i = 0; i < 6; i++){
20                _q.put(new Integer(i));
21                try{
22                    this.sleep(10);
23                }catch(InterruptedException ignored){ }
24            }
25        }
26
27        public void run(){
28            this.insertStuff();
29        }
30
31        public static void main(String[] args){
32            QTesterApp qt1 = new QTesterApp();
33            QTesterApp qt2 = new QTesterApp();
34            qt1.start();
35            qt2.start();
36        } // end main()
37    } // end QTesterApp class definition

```

Referring to example 17.10 — the `ActiveQueue` class gets its functionality from a `LinkedList`. The `LinkedList` class offers methods that allow you to insert objects at the beginning of the list and retrieve objects from the end of the list. The `Vector` reference `_listeners` will contain a collection of objects that have implemented the `QueueListenerInterface`. When a new element is added to an `ActiveQueue` object using the `put()` method on line 38, the `fireQueueInsertionEvent()` method is called. The `fireQueueInsertionEvent()` method, starting on line 54, uses an `Iterator` to step through the cloned `_listeners` `Vector`, calling the `queueInsertionPerformed()` method on each element.

Referring to example 17.12 — the `QTesterApp` class extends `Thread` and implements the `QueueListenerInterface`. It creates a static `ActiveQueue` object on line 6. This one static instance of `ActiveQueue` will be shared by all non-static instances of `QTesterApp`. The `run()` method beginning on line 27 simply calls the `insertStuff()` method, which inserts six `Integer` objects into the queue. The `main()` method starting on line 31 creates two `QTesterApp` instances and calls their `start()` methods. The results of running this program are shown in figure 17-20.



```

Terminal — tcsh
[rick@lillies-Computer:~/desktop/queueclass] swodog$ java QTesterApp
New object is in the queue!: 0
New object is in the queue!: 0
New object is in the queue!: 0
New object is in the queue!: 0
New object is in the queue!: 1
New object is in the queue!: 1
New object is in the queue!: 1
New object is in the queue!: 1
New object is in the queue!: 2
New object is in the queue!: 2
New object is in the queue!: 2
New object is in the queue!: 2
New object is in the queue!: 3
New object is in the queue!: 3
New object is in the queue!: 3
New object is in the queue!: 3
New object is in the queue!: 4
New object is in the queue!: 4
New object is in the queue!: 4
New object is in the queue!: 4
New object is in the queue!: 5
New object is in the queue!: 5
New object is in the queue!: 5
New object is in the queue!: 5
[rick@lillies-Computer:~/desktop/queueclass] swodog$

```

Figure 17-20: Results of Running Example 17.12

Quick Review

In Java collections framework up to and including 1.4.2 it is necessary to cast retrieved objects to their proper type if you need to call a method unique to that type. If you are calling methods that override `Object` class methods then casting is not required.

New data structures can be created from existing Java collections framework classes.

JAVA 5 Style Collections: Generics

The Java 5 platform introduced sweeping changes to the Java collections framework. The most important change was the introduction of Generics. Generics offer the ability to define the type of objects a collection contains when the collection is declared. Other changes include an enhanced for loop specifically designed to iterate over collections along with additional collection interfaces and classes.

JAVA 5 COLLECTION FRAMEWORK CORE INTERFACES

The Java 7 collections framework introduced additional core interfaces as shown in figure 17-21. The `<E>` to the right of each interface represents a type placeholder for the element type the collection will contain when it is created. For the `Map` and its sub interfaces the `<K, V>` represents the collection's `Key` and `Value` type placeholders. New inter-

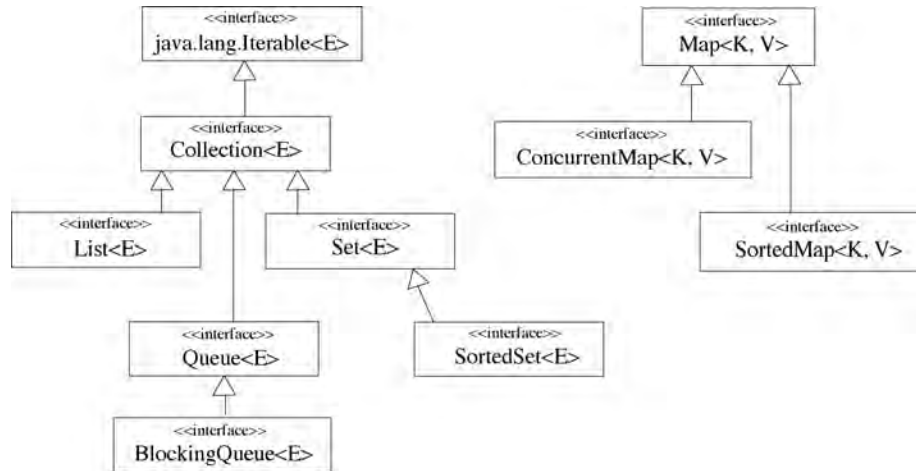


Figure 17-21: Java 5 Collections Framework Core Interface Hierarchy

faces now include Queue<E>, BlockingQueue<E>, and ConcurrentMap<K, V>. These new interfaces and their characteristics are listed and described in table 17-2.

Interface	Characteristic
Queue<E>	The Queue<E> interface represents a collection that holds elements prior to processing.
BlockingQueue<E>	A BlockingQueue<E> is a Queue<E> that is capacity bounded and can block on retrievals when the queue is empty or block on insertions when the queue is full.
ConcurrentMap<K, V>	The ConcurrentMap<K, V> interface is found in the java.util.concurrent package. It provides enhanced functionality in concurrent programming environments.

Table 17-2: New Java 5 Core Collection Interfaces

JAVA 5 COLLECTIONS FRAMEWORK SAMPLE PROGRAMS

This section offers several examples of the Java 5 collections framework in action.

SETTESTAPP PROGRAM REVISED

Let’s first take a look at a short program that uses generic collection classes. Example 17.13 offers a revised version of the SetTestApp class originally given in example 17.5.

17.13 SetTestApp.java
(generic version)

```

1      import java.util.*;
2
3      public class SetTestApp {
4          public static void main(String[] args){
5              List<Integer> list = new LinkedList<Integer>();
6              for(int i=0; i<50; i++){
7                  list.add(i % 10);
8              }
9              System.out.print("List contents: ");
10
11             for(Iterator i = list.iterator(); i.hasNext();){
12                 System.out.print(i.next());
13             }
14             System.out.println();
15
16             Set<Integer> set = new TreeSet<Integer>(list);
17             System.out.print("Set contents: ");
18             for(Iterator i = set.iterator(); i.hasNext();){
19                 System.out.print(i.next());

```

```

20     }
21
22     System.out.print("\nSet contents using for-each loop: ");
23     for(Integer i : set){
24         System.out.print(i);
25     }
26 } // end main()
27 } // end SetTestApp class definition

```

Referring to example 17.13 — this version of SetTestApp uses Java generic collection classes and the enhanced for loop (*for-each loop*) to iterate over one of the collections. It would be helpful to compare this version of the program with the Java 1.4.2 version during this discussion.

The first change appears on line 5 where the specific element type that will be contained in the list is specified in the angle brackets. In this example the LinkedList will store elements of type Integer. The for loop beginning on line 6 has been modified to generate only 50 int values. Notice now, however, that it is no longer necessary to explicitly create Integer objects with the new operator because the primitive int values are automatically boxed into Integer objects when they are inserted into the list on line 7. The for loop beginning on line 11 uses an Iterator to step through the list elements and print their values to the console. On line 16 the TreeSet object is created and its elements are specified to be of type Integer. The set elements are populated with the unique elements contained in the list as was done in the previous version of this program. The for loop beginning on line 18 again uses an Iterator to step through the set elements. An enhanced for loop is used on line 23 to step through the set elements one last time. Figure 17-22 gives the results of running this example.

```

C:\Documents and Settings\Rick Miller\Desktop\Projects\J5Set>java SetTestApp
List contents: 012345678901234567890123456789012345678901234567890123456789
Set contents: 0123456789
Set contents using for-each loop: 0123456789
C:\Documents and Settings\Rick Miller\Desktop\Projects\J5Set>_

```

Figure 17-22: Results of Running Example 17.13

WHEN TO USE THE ENHANCED FOR LOOP

You may rightly be wondering when is it appropriate to iterate over the elements of a collection using an enhanced for loop vs. an ordinary for loop and Iterator. An Iterator allows you to remove elements from a collection using the remove() method. If you simply want to step through a collection's elements without modification then you can use the enhanced for loop. An enhanced for loop effectively hides the Iterator. If, however, you need to remove elements from the collection when you're stepping through them then you must use an Iterator.

STATIC POLYMORPHISM – GENERIC METHODS

You can create generic methods with the help of Java 5 generics. A generic method is a method that can operate properly given different argument and/or return value types when the method is invoked.

17.14 GenericTest.java

```

1     import java.util.*;
2
3     public class GenericTest {
4
5         public static <T> void listArrayElements(T[] a){
6             for(T o : a){
7                 System.out.println(o);
8             }
9             System.out.println();
10        }
11
12        public static void main(String[] args){
13            Integer[] int_array = new Integer[10];
14            for(int i=0; i<int_array.length; i++){
15                int_array[i] = i;
16            }
17        }

```

```

18
19     GenericTest.listArrayElements(int_array);
20
21     Person[] person_array = new Person[5];
22
23     person_array[0] = new Person("Rick", "W", "Miller", 1966, 8, 28, Person.MALE);
24     person_array[1] = new Person("Steve", "J", "Jones", 1986, 2, 10, Person.MALE);
25     person_array[2] = new Person("Howard", "Josephus", "Stern", 1972, 5, 5, Person.MALE);
26     person_array[3] = new Person("Sally", "Sue", "Smith", 1987, 7, 2, Person.FEMALE);
27     person_array[4] = new Person("Karen", "H", "Stevens", 1977, 3, 18, Person.FEMALE);
28
29     GenericTest.listArrayElements(person_array);
30 } // end main()
31 } // end GenericTest class definition

```

Referring to example 17.14 — on line 5 the `GenericTest` class defines a static method named `listArrayElements()`. The `<T>` denotes a generic type placeholder. The `T` is then used in the method definition to declare an array parameter and to specify the type of element in the enhanced for loop on line 6. In the `main()` method beginning on line 12 an array of `Integers` is created and initialized with the help of the for loop on line 14. On line 19 the contents of `int_array` are printed to the console with the help of the `listArrayElements()` method. Next, on line 21, an array of `Person` references is created. Each element of `person_array` is initialized on lines 23 through 27 and the `listArrayElements()` method is used again on line 29 to print the contents of `person_array` to the console. The results of running this program are shown in figure 17-23.

```

C:\Documents and Settings\Rick Miller\Desktop\Projects\GenericTest>java GenericTest
0
1
2
3
4
5
6
7
8
9
Rick W Miller Male 39
Steve J Jones Male 19
Howard Josephus Stern Male 33
Sally Sue Smith Female 18
Karen H Stevens Female 28
C:\Documents and Settings\Rick Miller\Desktop\Projects\GenericTest>_

```

Figure 17-23: Results of Running Example 17.14

Quick Review

The Java 5 platform introduced generics to the Java collections framework. Generic collections lets you specify element type in angle brackets.

Primitive type values will be autoboxed into their corresponding wrapper classes when being inserted into a generic collection. They will be unboxed when accessed.

The enhanced for loop hides the `Iterator` when you step through a collection's elements. If you need to remove an element from a collection during iteration then you must use an ordinary for loop and `Iterator` combination.

Moving Forward

This section only provided a brief glimpse of the power and versatility of the Java 5 collections framework and Java generics. For more information on programming with generics please consult one of the excellent references on generics listed at the end of the chapter.

SUMMARY

The purpose of the Java collections framework is to provide Java programmers with a unified, comprehensive way to manipulate collections of objects in their programs across deployment platforms. The Java collections framework is organized into four primary areas: 1) core collection interfaces, 2) general purpose implementation classes, 3) algorithms, and 4) infrastructure.

The core collection interfaces consist of `Collection`, `List`, `Set`, `SortedSet`, `Map`, and `SortedMap`. `Map` and `SortedMap` are not collections but have methods that return their elements in the form of a `Collection` object. The general purpose implementation classes employ arrays, linked lists, hashables, and red-black trees as their underlying data structures.

Array elements can be quickly retrieved but insertions into the array at element positions other than the end can exact a performance penalty because element values to the right of the insertion must be copied to the right.

Linked lists overcome the problems associated with array element insertion, however, the linked list must be traversed to find a desired element. List traversal exacts a performance penalty. Hashables are a cross between an array and a linked list.

Hashtable element storage locations are assigned based on the results of a hash function. Hashtable collisions are resolved by linking together elements that share the same hashtable element location. Hashables can save storage space but a poor hash function may result in frequent hashtable collisions which will result in a list traversal performance penalty.

Red-black trees store their elements in sorted order upon insertion. Element insertion times grow with tree size. However, tree element access time is good overall regardless of tree size.

The `Collections` class contains static methods that implement collection manipulation algorithms. The `Arrays` class contains static methods that can be used to manipulate arrays. A collection's elements can be converted into an array and in this form can be manipulated by the static methods of the `Arrays` class.

Infrastructure classes and interfaces are employed across the entire collections framework.

In Java collections framework up to and including 1.4.2 it is necessary to cast retrieved objects to their proper type if you need to call a method unique to that type. If you are calling methods that override `Object` class methods then casting is not required.

New data structures can be created from existing Java collections framework classes.

The Java 5 platform introduced generics to the Java collections framework. Generic collections lets you specify element type in angle brackets.

Primitive type values will be autoboxed into their corresponding wrapper classes when being inserted into a generic collection. They will be unboxed when accessed.

The enhanced for loop hides the `Iterator` when you step through a collection's elements. If you need to remove an element from a collection during iteration then you must use an ordinary for loop and `Iterator` combination.

Skill-Building Exercises

- API Research:** Research the core collection interfaces provided by the Java platform version 1.4.2 collections framework API. Pay particular attention to the methods published by each collection interface and subinterface. Note the optional methods.
- API Research:** Research the general purpose implementation classes provided by the Java platform version 1.4.2 collections framework. Answer the following questions for each class:
 - Does the collection extend an abstract class, implement a core collection interface, or both?
 - What methods, if any, does the collection class implement in addition to those of its base class or interface?
 - Describe the general performance characteristics of the collection class's underlying data structure.
- API Research:** Research the core collection interfaces provided by the Java platform version 5 collections frame-

work API. Pay particular attention to the methods published by each collection interface and subinterface. Note the optional methods.

4. **API Research:** Research the general purpose implementation classes provided by the Java platform version 5 collections framework. Answer the following questions for each class:
 - a) Does the collection extend an abstract class, implement a core collection interface, or both?
 - b) What methods, if any, does the collection class implement in addition to those of its base class or interface?
 - c) Describe the general performance characteristics of the collection class's underlying data structure.
5. **Research Generics:** Obtain and study Gilad Bracha's paper titled *Generics in the Java Programming Language*. You should be able to find this paper easily on the Web.
6. **Programming:** Write several short programs that utilize some of the collections classes not formally discussed in this chapter.
7. **Algorithms:** Study the static methods supplied by the Collections class. List each method and write a brief description of its functionality.

SUGGESTED PROJECTS

1. **Employee Management Program:** Using the Person class, Payable interface and Employee, HourlyEmployee, and SalariedEmployee classes originally presented in chapter 11 write a GUI-based employee management program that lets users create and pay employees. Maintain employee objects in a collection. If you use Java 5 maintain employee objects in a generic collection whose elements are of type Employee.
2. **Implement The Comparator Interface:** *Background* — Objects inserted into an ordered collection must implement the `java.lang.Comparable` or the `java.util.Comparator` interface. The `Comparable` interface is used to order objects according to their natural ordering. (*For example, Integers have a natural ordering.*) The `Comparator` interface is used when objects do not have a natural ordering as defined by `Comparable` or when you want to order them differently from their natural ordering. *Task* — Modify the Person class to implement the `Comparator` interface. Have it compare Person objects according to their age. Write a program that creates several different Person objects and insert them into an ordered collection in unsorted order. Print the contents of the collection to the console.
3. **Aircraft Engine Simulation Revisited:** Return to chapter 11 and reexamine the aircraft engine simulation code. In its current form, the Part class contains and manages a `HashTable` of subparts. Upgrade the simulation to utilize a Java 5 generic `Hashtable`.

SELF-TEST QUESTIONS

1. What's the purpose of the Java collections framework?
2. Into what four primary areas is the Java collections framework organized?
3. What are the six core collection interfaces offered in the Java platform version 1.4.2 collections framework?
4. Describe the performance characteristics of a linked list.

5. Describe the performance characteristics of a hashtable.
6. Describe the performance characteristics of a red-black tree.
7. What's the purpose of an Iterator?
8. Describe the fundamental differences between the Java 5 collections framework and previous collections framework versions.
9. What's the difference between an Iterator and a ListIterator?
10. When would you chose to iterate over the elements of a Java 5 collection using the enhanced for loop vs. an ordinary for loop and an Iterator?

REFERENCES

Gilad Bracha. *Generics in the Java Programming Language*. 5 July 2004.

John Franco. *Red-Black Tree Demonstration Applet*. [<http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html>]

java.util.TreeSet.java Source Code Listing, J2SE Version 1.4.2_05

java.util.Hashtable.java Source Code Listing, J2SE Version 1.4.2_05

java.util.TreeMap.java Source Code Listing, J2SE Version 1.4.2_05

Thomas H. Cormen, et. al. *Introduction To Algorithms*. The MIT Press, Cambridge, MA. ISBN: 0-262-03141-8

Donald E. Knuth. *The Art of Computer Programming, Volume 1, Fundamental Algorithms, Third Edition*. Addison-Wesley, Reading, MA. ISBN: 0-201-89685-0

Donald E. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching, Second Edition*. Addison-Wesley, Reading, MA. ISBN: 0-201-89685-0

NOTES

CHAPTER 18



REX STEELE

File I/O

LEARNING OBJECTIVES

- *DEMONSTRATE YOUR ability TO CONDUCT file I/O OPERATIONS*
- *LIST AND describe THE PRIMARY JAVA CLASSES REQUIRED TO CONDUCT file I/O OPERATIONS*
- *DESCRIBE THE BEHAVIOR OF A SERIALIZED object WHEN SAVED TO A file*
- *LIST THE STEPS REQUIRED TO OPEN A file, WRITE objects TO THE file, READ FROM A file, AND close THE file*
- *DEMONSTRATE YOUR ability TO READ byte STREAMS FROM A file*
- *DEMONSTRATE YOUR ability TO WRITE byte STREAMS TO A file*
- *DEMONSTRATE YOUR ability TO READ CHARACTER STREAMS FROM A file*
- *DEMONSTRATE YOUR ability TO WRITE CHARACTER STREAMS TO A file*
- *DEMONSTRATE YOUR ability TO READ AND WRITE fixed-length file RECORDS*
- *DEMONSTRATE YOUR ability TO PROPERLY HANDLE EXCEPTIONS RELATED TO file I/O OPERATIONS*
- *DEMONSTRATE YOUR ability TO CONDUCT RANDOM file I/O OPERATIONS*

INTRODUCTION

A thorough understanding of file input and output programming is an essential skill you must possess to create complex programs that store and retrieve object or program state information to and from auxiliary storage devices. Luckily, Java makes file I/O programming easy with the classes provided in the `java.io` package.

The purpose of this chapter is to show you how to write and read bytes, characters, Java primitive types, and objects to and from files. Along the way you will learn how to select and use the appropriate `java.io` classes for the file I/O task at hand.

This chapter starts with an overview of the primary classes of the `java.io` package. Although I will not show you how to use every class in the `java.io` package you will have a good understanding of how the `java.io` package is organized when you complete this chapter. This knowledge will make it easy for you to find the right `java.io` class when the need arises.

The primary programming example will show you how to write a Java adapter that accesses a legacy application flat-file data store. The example will demonstrate the use of the `RandomAccessFile` class in the manipulation of fixed-length record fields.

To understand the material in this chapter you should be thoroughly familiar with the concepts presented in chapters 1 through 11, 15, & 16.

Although the focus of this chapter is on using the `java.io` package to perform file I/O, everything you learn here regarding the use of `InputStreams`, `OutputStreams`, `Readers`, and `Writers`, can be applied when you need to perform network-based I/O.

JAVA I/O PACKAGE OVERVIEW

The `java.io` package contains numerous classes to help you perform input and output (I/O) operations on a wide variety of sinks and sources including arrays, strings, processes, and files. In the context of I/O operations the term *sink* describes a destination to which data is written while the term *source* indicates an origination point from which data is read. This chapter focuses on using a subset of the `java.io` classes to access files. An understanding of the use of these I/O classes will lead to an understanding of how to use the other types of sinks and sources.

At first approach the `java.io` package can be intimidating to novice Java programmers. There are `InputStreams` and `OutputStreams`, `Readers`, and `Writers`, and a bewildering array of subclasses that inherit from these classes. To help tame the conceptual complexity it is helpful to group and categorize the `java.io` package classes in a couple of different ways. One method of grouping them is by inheritance hierarchy as is shown in figure 18-1. An inheritance hierarchy grouping of the `java.io` package classes offers several clues about the intended use of each class.

Referring to figure 18-1 — an inheritance hierarchy grouping yields six distinct class categories: 1) the `File` class, 2) `InputStreams`, 3) `OutputStreams`, 4) `Readers`, 5) `Writers`, and 6) the `RandomAccessFile`. Each class category is discussed in greater detail below.

File Class

The `File` class is used to manipulate information about a file rather than the data contained within a file. A `File` object represents a file at a metadata level. Once you create a `File` object you can use it to obtain information about a file such as its length, its read/write ability, and other attributes. You can also use a `File` object to create an empty file, to delete a file, to create or delete directories, or manipulate lists of files and directories. You do not need to use a `File` object to create or use a file for data I/O, however, in many file I/O programming situations you will find its services extremely helpful.

INPUTSTREAM CLASSES

The `InputStream` category includes classes that inherit from the `InputStream` class. `InputStream` classes treat their data source as a stream of bytes. A byte stream is simply a sequence of bytes that could represent any type of stored

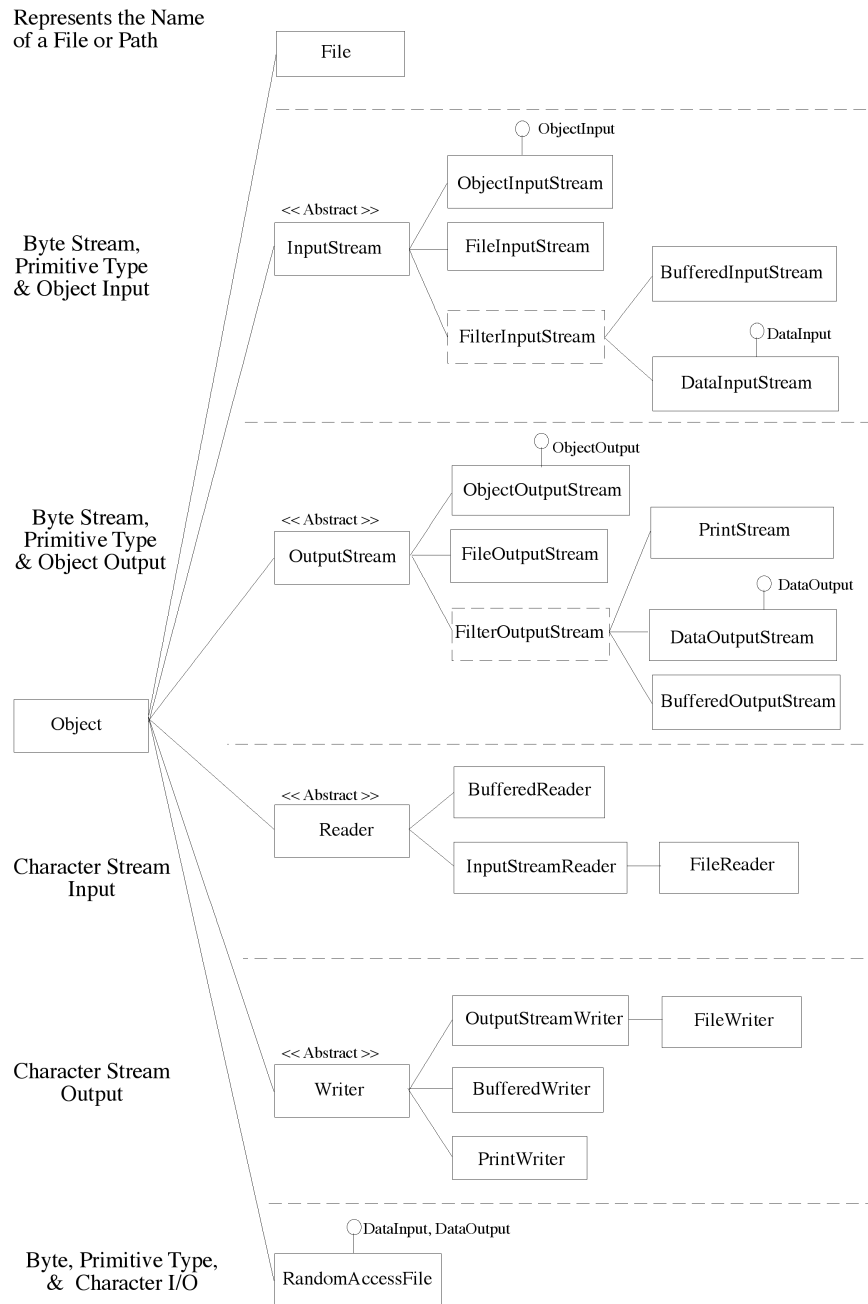


Figure 18-1: Partial java.io Package Hierarchy

data. The `InputStream` class itself is abstract. This means you must use one, or a combination, of its subclasses to do any real work.

`InputStream` classes can be used to read raw bytes from a file as is the case with the `FileInputStream` class, or they can be used to read Java primitive data types (*DataInputStream*), or serialized objects (*ObjectInputStream*). You can also read characters but not as efficiently or as flexibly as you can with a `Reader`. Characters read with an `InputStream` class will be either 8-bit ASCII or UTF-8 (*an ASCII-compatible encoding of a Unicode character*). Regardless of what is being read, if it is being read with the help of an `InputStream` class its data source is treated as a stream of bytes.

OUTPUTSTREAM CLASSES

The `OutputStream` category is the complement of its `InputStream` cousin. `OutputStream` classes write data to a file as a sequence of bytes. You can use an `OutputStream` to write raw bytes, primitive Java data types, and serialized objects. You can write characters with `OutputStreams` as well but not as efficiently or as flexibly as you can with a `Writer` class. Like its `InputStream` counterpart, the `OutputStream` class is abstract so you must use one, or a combination, of its subclasses to do any real work.

READER CLASSES

The `Reader` class category enables you to read a sequence of Unicode characters from a file. You can specify the character encoding scheme to use via a constructor call when you create an `InputStreamReader` object. Although the underlying data is still stored as a series of bytes, the difference between a `Reader` and an `InputStream` is the capability to specify the way in which the bytes are translated into characters vs. being locked into one specific encoding.

The `Reader` class is abstract. You must use one, or a combination, of its subclasses to perform any real work.

WRITER CLASSES

The `Writer` class enables you to write a sequence of Unicode characters to a file. You can specify the character encoding scheme to use via a constructor call when you create an `OutputStreamWriter` object. The `Writer` class is abstract so you must use one, or a combination, of its subclasses to perform any real work.

RANDOMACCESSFILE CLASS

The `RandomAccessFile` class stands on its own as it is not an `InputStream`, `OutputStream`, `Reader`, or `Writer`. The `RandomAccessFile` class enables you to perform random access file input and output operations as opposed to sequential file I/O offered by the byte stream and character stream classes. You can use a `RandomAccessFile` object to write bytes, characters, and Java primitive types. If, however, you want to read and write serialized objects, you'll need to use the `ObjectInputStream` and `ObjectOutputStream` classes.

HOW DO YOU CHOOSE BETWEEN BYTE STREAMS, CHARACTER STREAMS, AND RANDOMACCESSFILE?

After reading the previous descriptions of `InputStreams`, `OutputStreams`, `Readers`, `Writers`, and The `RandomAccessFile` class you may be asking yourself, “OK, but which ones should I use in my program? It seems like `RandomAccessFile` does everything I need!” Good question.

Generally speaking, if you need to process large files of raw byte information such as image files, music files, or other similar types of data, use the byte stream classes that inherit from `InputStream` and `OutputStream`. You would also use these files if you were serializing objects or collections of objects. (*i.e.*, *saving a `Vector` or `HashMap` of `People` objects to a file.*) Any operation that calls for sequentially accessing raw byte data from a file, or the serialization and deserialization of objects — use the byte stream classes.

If, on the other hand, you were working exclusively with large amounts of text data then the `Reader` and `Writer` classes will prove more efficient. These classes are especially helpful if you have a need to internationalize your program as they deal more effectively with Unicode characters than do the byte stream classes.

Finally, if you need to store and retrieve a combination of data such as primitive types mixed with text in a non-sequential manner then the `RandomAccessFile` is your ticket.

ANOTHER WAY TO CATEGORIZE THE JAVA I/O CLASSES

Another helpful way to categorize the Java I/O classes is by what type of object is required in their constructors. This will help you figure out what class, or combination of classes, can be used to solve your particular file I/O problem. Table 18-1 provides an organization of the Java I/O classes by constructor argument type.

Constructor Argument Type	Input & Output Streams	Readers & Writer	Other
Classes that require a <i>File</i> or <i>String</i> reference that represents the name of the file in the constructor	FileInputStream FileOutputStream	FileReader FileWriter	RandomAccessFile File
Classes that require an <i>InputStream</i> reference in the constructor	BufferedInputStream DataInputStream ObjectInputStream <i>FilterInputStream</i> †	InputStreamReader	
Classes that require an <i>OutputStream</i> reference in the constructor	BufferedOutputStream DataOutputStream ObjectOutputStream PrintStream <i>FilterOutputStream</i> †	OutputStreamWriter	
Classes that require a <i>Reader</i> reference in the constructor		BufferedReader	
Classes that require a <i>Writer</i> reference in the constructor		BufferedWriter	
Classes that require either an <i>OutputStream</i> or a <i>Writer</i> reference in the constructor		PrintWriter	
† - The <i>FilterInputStream</i> and <i>FilterOutputStream</i> classes have protected constructors and must be subclassed to provide functionality.			

Table 18-1: Java File I/O Classes By Constructor Argument Type

Referring to table 18-1 — With the exception of the `File` class, all the classes appearing in the first row of the table can be considered to be the primary file I/O classes. Each of the classes `FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`, and `RandomAccessFile` take either a `File` or `String` object as constructor arguments. These `File` or `String` arguments represent the name of the file to open or create. I will label these classes as being *file-terminal classes*, meaning that to perform I/O on a file you must create one or more objects of these types.

The `RandomAccessFile` class is used by itself to perform both file input and output. It has an extensive user interface that lets you write and read bytes, characters, and primitive types. Because of its extensive set of interface methods I will label this class as being a *user-fronting class*. Because the `RandomAccessFile` class is both a file-terminal class and user-fronting class it is used stand-alone to perform file I/O operations and needs no help from other `java.io` classes to perform its job.

The `FileInputStream`, `FileOutputStream`, `FileReader`, and `FileWriter` classes have only a few interface methods which are designed to read streams of bytes or characters. In fact, the `FileReader` and `FileWriter` classes possess only constructors and it is the methods provided by their base classes `InputStreamReader` and `OutputStreamWriter` respectively that must be considered when using these classes. Because their interfaces are so sparse these classes are most often used in concert with other stream and reader classes to provide sufficient programmer functionality. Because of this I consider these classes, and by relationship the `InputStreamReader` and `OutputStreamWriter` classes as well, not to be user-fronting classes.

The remaining classes in table 18-1 are considered to be either *intermediate* or user-fronting classes. An intermediate class is one that is used in conjunction with a file-terminal class to provide augmented functionality but does not possess sufficient user interface methods to make it a user-fronting class. Any class name that includes the word `Buffer` is considered to be an intermediate class. The purpose of a class that contains the word `Buffer` is to improve the efficiency of file I/O operations through the use of *I/O buffering*. Buffering improves I/O efficiency because individ-

ual disk reads and writes are temporarily stored in memory (*a buffer, usually in the form of an internal array*) until the contents of the buffer is *flushed* resulting in a combined disk read or write operation.

Table 18-2 provides a grouping of classes according to their classification as a file terminal, intermediate, or user fronting class.

Usage Characteristic	InputStream & Output Streams	Readers & Writers	Other
File-Terminal	FileInputStream FileOutputStream	FileReader FileWriter	RandomAccessFile
Intermediate	BufferedInputStream BufferedOutputStream	InputStreamReader BufferedReader OutputStreamWriter BufferedWriter	
User-Fronting	DataInputStream ObjectInputStream DataOutputStream ObjectOutputStream PrintStream	PrintWriter	RandomAccessFile

Table 18-2: Java I/O Classes Organized By File-Terminal, Intermediate, Or User-Fronting Characteristic

Now, before I get into trouble with any Java experts reading this chapter, I have to clarify a few things regarding InputStreams, OutputStreams, Readers, and Writers. Just because I have labeled a class as being intermediate or file-terminal does not mean it cannot be used directly for file I/O. If you need to write or read arrays of bytes or chars to or from a file then the interface methods provided by these classes will suffice. However, the user-fronting classes provide a wide assortment of interface methods which are more convenient to programmers.

Referring to table 18-2 — this arrangement of classes is almost like a menu. Say, for example, you want to write primitive data types to a file using a byte stream. You will need a file-terminal object, optionally an intermediate object, and a user-fronting object. Browsing the table you might select the following: FileOutputStream, BufferedOutputStream, and DataOutputStream. If you wanted to write characters you could go the Writer path by selecting FileWriter, BufferedWriter, and PrintWriter.

The following sections will show you how to use the java.io classes in various combinations to conduct file I/O operations.

Quick Review

The classes in the java.io package can be used to perform I/O operations on various types of data sinks and sources. A sink is a destination to which data is written, and a source is an origination point from which data is read. Strings, arrays, processes, and files are examples of the types of sinks and sources supported by the java.io classes. This chapter focuses on using the java.io classes to perform I/O operations on files.

The java.io package can be intimidating to novice Java programmers. But there are several ways to organize the classes in the java.io package to help tame the conceptual complexity. A class hierarchy organization gives valuable clues regarding the intended use of the java.io classes.

The File class represents the name of a file or directory and is used to manipulate file metadata.

The InputStream and OutputStream classes operate on byte streams. InputStreams and OutputStreams enable you to read and write bytes, primitive types, 8-bit characters, and serialized objects. When performing character I/O with byte streams the local default character encoding is utilized.

The Reader and Writer classes operate on 16-bit Unicode characters. The character encoding can be specified. This is an important internationalization feature.

The `RandomAccessFile` class is used to perform both file input and output on bytes, chars, and primitive types.

It is also helpful to categorize `java.io` classes into file-terminal, intermediate, and user-fronting categories. File-terminal classes are used to create or open a file for input, output, or input/output operations. There are five file-terminal classes: `FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`, and `RandomAccessFile`.

Intermediate classes are used to enhance the performance of file-terminal classes. Generally speaking, any class that contains the word `Buffer` is an intermediate class.

User-fronting classes are those classes that have a wide variety of user interface methods that make byte, character, primitive type, and/or object I/O operations convenient for the programmer.

Using The File Class

The `File` class represents the name of a file or path. It is used to manipulate file metadata information such as the filename, directory listing, file length, whether the filename is a directory or a file, whether the file is readable, writable, or both, etc.

You do not need to use a `File` object to create a file for I/O operations because all the file terminal class constructors can take a `String` as well as a `File` as a argument. However, at times you will need the functionality provided by the `File` class so it's important that you understand its operation.

Example 18.1 gives a short program that creates a `File` object and calls a few of its methods.

18.1 FileClassTesterApp.java

```

1      import java.io.*;
2
3      public class FileClassTesterApp {
4          public static void main(String[] args){
5              File file = null;
6              try{
7                  file = new File(args[0]);
8                  System.out.println(file.getName());
9                  System.out.println(file.getPath());
10                 System.out.println(file.getAbsolutePath());
11                 System.out.println(file.getCanonicalPath());
12                 System.out.println(file.isDirectory());
13             }catch(Exception ignored){ }
14         }
15     }
16 }

```

Referring to example 18.1 — this program takes the text string entered on the command line and creates a `File` object having that name. The `File` reference is declared on line 5 and the `File` object is created on line 7. Various methods are called on the `File` object on lines 8 through 12.

It's important to note that although a `File` object is created, the actual file is not created. This is demonstrated by running the example. The results of running this program are shown in figure 18-2.

```

Terminal — tcsh
[rick-millers-computer:~/desktop/fileclasstester] swodog% ls
FileClassTesterApp.class  FileClassTesterApp.java
[rick-millers-computer:~/desktop/fileclasstester] swodog% java FileClassTesterApp TestFile.dat
TestFile.dat
TestFile.dat
/Users/swodog/Desktop/FileClassTester/TestFile.dat
/Users/swodog/Desktop/FileClassTester/TestFile.dat
false
[rick-millers-computer:~/desktop/fileclasstester] swodog% ls
FileClassTesterApp.class  FileClassTesterApp.java
[rick-millers-computer:~/desktop/fileclasstester] swodog% █

```

Figure 18-2: Results of Running Example 18.1

Referring to figure 18-2 — notice that before the program was executed an `ls` (*list directory*) was performed which showed there were two files in the directory: `FileClassTesterApp.class` and `FileClassTesterApp.java`. During program execution the `File` object is created and a few of its methods called. When the program completes another `ls` is performed and yields the same two classes in the directory.

You can create a file with a File object by calling its `createNewFile()` method but creating a file in this manner results in an empty file.

Example 18.2 shows how you can use the File class in conjunction with the JFileChooser swing component to open or create a new File object.

18.2 JFileChooserTestApp.java

```

1      import javax.swing.*;
2      import java.io.*;
3
4      public class JFileChooserTestApp {
5          public static void main(String[] args){
6              int    chooser_action;
7              File  file;
8              JFileChooser file_chooser = new JFileChooser();
9              chooser_action = file_chooser.showOpenDialog(new JFrame());
10             if(chooser_action == JFileChooser.APPROVE_OPTION){
11                 file = file_chooser.getSelectedFile();
12                 System.out.println(file.getName());
13                 System.out.println(file.getPath());
14                 System.out.println(file.length());
15                 System.exit(0);
16             }else{
17                 System.exit(0);
18             }
19         }
20     }

```

Referring to example 18.2 — a JFileChooser object is created on line 8 and is used on lines 10 and 11 to choose an existing file or to create a new file. When the file is selected the file reference is initialized on line 11 by a call to the JFileChooser `getSelectedFile()` method. Several methods are called via the file reference on lines 12 through 14 and then the program exits. Figure 18-3 shows the results of running this program.

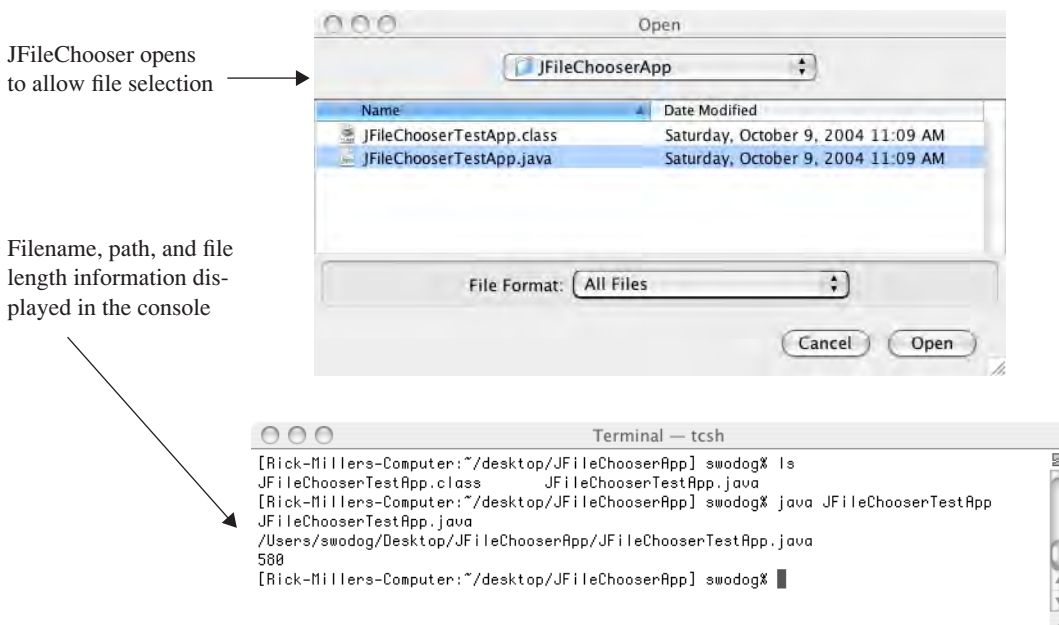


Figure 18-3: Results of Running Example 18.2

Referring to figure 18-3 — upon application execution the JFileChooser dialog opens and allows a file to be selected. In this example the source file `JFileChooserTestApp.java` is selected and the Open button is clicked. This results in the file's information being written to the console.

Quick Review

The File class is used to manipulate file metadata such as a file's name, length, and path. The creation of a File object does not result in an actual file being created on disk, however, you can create an empty file by calling the createNewFile() method.

SEQUENTIAL BYTE STREAM FILE I/O USING INPUT- & OUTPUTSTREAMS

InputStreams and OutputStreams are used to perform sequential byte-oriented I/O. Refer to figure 18-1 again and study the InputStream and OutputStream inheritance hierarchy. The InputStream and OutputStream classes are abstract so you must use one or more of their subclasses, alone or in combination with each other, to perform file I/O operations. When you perform file I/O operations with InputStreams and OutputStreams you read and write data sequentially. You cannot maneuver around the file like you can with the RandomAccessFile class.

I will discuss the OutputStream classes first and show you how to write byte streams and objects to a file. I will then show you how to read those files using InputStream classes.

OUTPUTSTREAMS

The FileOutputStream, BufferedOutputStream, DataOutputStream, ObjectOutputStream, and PrintStream classes are used to write byte streams, primitive types, objects, and text to disk. The use of each class is discussed and demonstrated below.

FileOutputStream

You can use the FileOutputStream class to perform sequential byte output to a file. The FileOutputStream user interface is minimal. In addition to five overloaded constructors it provides three overloaded versions of a write() method you can use to write a single byte, an array of bytes, or a segment of an array of bytes to a file. It provides a close() method which should be called to close the file when you have finished writing data. Its two other methods getChannel() and getFD() will not be discussed here. (*The getChannel() and getFD() methods are used in conjunction with classes belonging to the java.nio package.*)

Example 18.3 shows the OutputStream class in action.

18.3 FOS_TesterApp.java

```

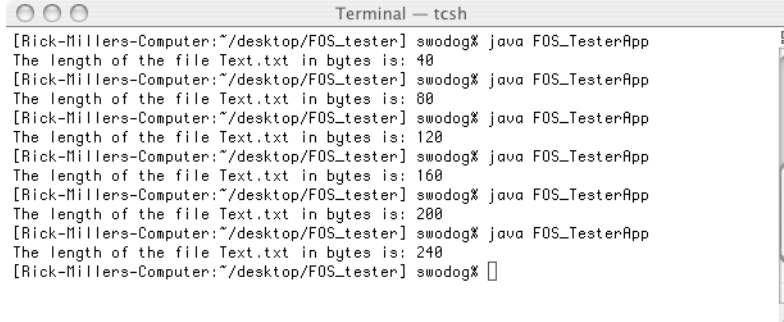
1      import java.io.*;
2
3      public class FOS_TesterApp {
4          public static void main(String[] args){
5              FileOutputStream fos;
6              File file;
7              try{
8                  fos = new FileOutputStream("Test.txt", true);
9                  if(args.length != 0){
10                     fos.write(args[0].getBytes());
11                 }else{
12                     fos.write(("Ohhhh do you love Java like I love Java?").getBytes());
13                 }
14                 fos.close();
15                 file = new File("test.txt");
16                 System.out.println("The length of the file Text.txt in bytes is: " + file.length());
17             }catch(Exception ignored){ }
18         } // end main
19     } // end class

```

Referring to example 18.3 — the program will write to disk any text entered on the command line (*up to the first space*) when the program is executed. If no text is entered on the command line then it writes the phrase “Ohhhh do you love Java like I love Java?” to disk. The FileOutputStream object is created on line 8. The arguments to the constructor include the name of the file to open and the boolean literal true which enables file appending.

Notice how the String.getBytes() method is called on the String literal on line 12. The close() method is called on line 14 to close the file. After the file is closed a File object is created and used to get the length of the test.txt file. If

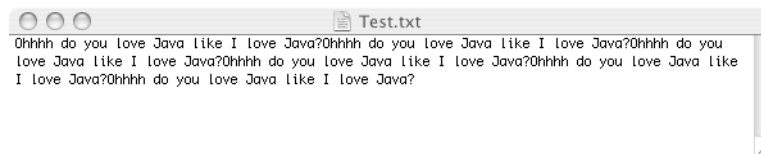
you run this program repeatedly without deleting the file in between runs it will append the new text to the end of the file. Figure 18-4 shows the results of running this program several times.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/FOS_tester] swodog% java FOS_TesterApp
The length of the file Text.txt in bytes is: 40
[Rick-Millers-Computer:~/desktop/FOS_tester] swodog% java FOS_TesterApp
The length of the file Text.txt in bytes is: 80
[Rick-Millers-Computer:~/desktop/FOS_tester] swodog% java FOS_TesterApp
The length of the file Text.txt in bytes is: 120
[Rick-Millers-Computer:~/desktop/FOS_tester] swodog% java FOS_TesterApp
The length of the file Text.txt in bytes is: 160
[Rick-Millers-Computer:~/desktop/FOS_tester] swodog% java FOS_TesterApp
The length of the file Text.txt in bytes is: 200
[Rick-Millers-Computer:~/desktop/FOS_tester] swodog% java FOS_TesterApp
The length of the file Text.txt in bytes is: 240
[Rick-Millers-Computer:~/desktop/FOS_tester] swodog% █
```

Figure 18-4: Results of Running Example 18.3

Referring to figure 18-4 — notice how the file length grows each time the application executes. Figure 18-5 shows how the file looks when opened in a text editor. Notice how each successive write appended text to the end of the file.



```
Test.txt
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?Ohhhh do you
love Java like I love Java?Ohhhh do you love Java like I love Java?Ohhhh do you love Java like
I love Java?Ohhhh do you love Java like I love Java?
```

Figure 18-5: Contents of test.txt After Executing Example 18.3 Six Times

If you want to overwrite the file you can call the constructor on line 8 with the boolean literal *false* or use the *FileOutputStream(String name)* version of the constructor which, by default, sets the append property to false.

BufferedOutputStream

When using the *FileOutputStream* class to perform file output, each time the *write()* method is called a disk write occurs. In situations where you anticipate a significant amount of disk writes you may want to improve your program's I/O efficiency by using the *BufferedOutputStream* class. The *BufferedOutputStream* class stores data writes in an internal memory array (*buffer*). The data is actually written to disk when one of two events occur: 1) when the buffer fills up, or 2) when the *flush()* method is called to force a write.

To use a *BufferedOutputStream* object you first create the *FileOutputStream* object then use it to create the *BufferedOutputStream* object. You then call the *write()* method on the *BufferedOutputStream* object. Example 18.4 shows the *BufferedOutputStream* class in action.

18.4 *BOS_TesterApp.java*

```
1      import java.io.*;
2
3      public class BOS_TesterApp {
4          public static void main(String[] args){
5              FileOutputStream fos;
6              BufferedOutputStream bos;
7              File file;
8              try{
9                  file = new File("test.txt");
10                 fos = new FileOutputStream(file);
11                 bos = new BufferedOutputStream(fos); // default buffer size is 512 bytes
12                 while(file.length() == 0){
13                     if(args.length != 0){
14                         bos.write(args[0].getBytes());
15                     }else{
16                         bos.write(("Ohhhh do you love Java like I love Java?").getBytes());
17                     }
18                     System.out.println("The length of the file Text.txt in bytes is: " + file.length());
19                 }
20                 fos.close();
21             }catch(Exception ignored){ }
22         } // end main
```

```
23     } // end class
```

Referring to example 18.4 — this program looks similar to example 18.3 but with a few important changes. First, a File object is created on line 9 and is used to create the FileOutputStream on line 10. The fos reference is passed as an argument to the BufferedOutputStream constructor on line 11 to create the BufferedOutputStream object. A while loop is used starting on line 12 to repeatedly write to the BufferedOutputStream buffer. The while loop will repeat until data is actually written to the file. (*i.e.*, *The buffer fills up at which time its contents are written to the disk in one stroke.*) Figure 18-6 shows the results of running this program. Figure 18-7 shows the contents of the test.txt file after the application execution completes.

```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/bos_tester] swodog% java B05_TesterApp
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 480
[Rick-Millers-Computer:~/desktop/bos_tester] swodog%
```

Figure 18-6: Results of Running Example 18.4

```
test.txt
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
```

Figure 18-7: Contents of test.txt File After Executing Example 18.4

The default buffer size for a BufferedOutputStream object is set to 512 bytes. If the next write operation performed on the BufferedOutputStream object will exceed the buffer size then the buffer's contents are written to disk automatically. You can force the write by calling the flush() method if required. You can also set the internal buffer size via the constructor when you create a BufferedOutputStream object.

DATAOUTPUTSTREAM

The DataOutputStream is used to write streams of bytes as well as primitive type byte values to a file. If it is used in concert with a FileOutputStream object it provides no buffering and every write will be immediately written to disk. To provide write buffering use the DataOutputStream in conjunction with a BufferedOutputStream object. Example 18.5 shows the DataOutputStream class in action.

18.5 DOS_TesterApp.java

```
1     import java.io.*;
2
3     public class DOS_TesterApp {
4         public static void main(String[] args){
5             FileOutputStream fos;
6             BufferedOutputStream bos;
7             DataOutputStream dos;
8             File file;
9             try{
10                file = new File("test.txt");
11                fos = new FileOutputStream(file);
12                bos = new BufferedOutputStream(fos); // default buffer size is 512 bytes
13                dos = new DataOutputStream(bos);
14                while(file.length() == 0){
15                    if(args.length != 0){
16                        dos.writeBytes(args[0]);
17                    }else{
18                        dos.writeBytes("Ohhhh do you love Java like I love Java?");
```

```

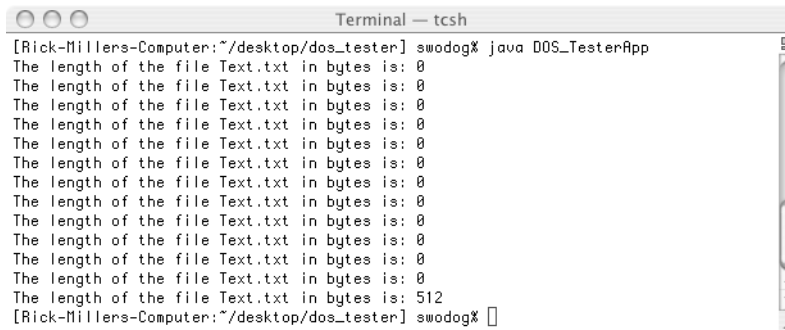
19     }
20     System.out.println("The length of the file Text.txt in bytes is: " + file.length());
21     }
22     fos.close();
23     }catch(Exception ignored){ }
24     } // end main
25     } // end class

```

Referring to example 18.5 — the File object is created on line 10. The file reference is used to create the FileOutputStream object on line 11. The fos reference is then used to create the BufferedOutputStream object on line 12, and the bos reference is used to create the DataOutputStream object on line 13. There is sort of a rhythm to it. First create the File, then the FileOutputStream, then the BufferedOutputStream, then the DataOutputStream. (*The File object is optional.*)

The dos reference is then used in the program to perform the writes to the file as is done on lines 16 and 18 with the writeBytes() method. The writeBytes() method takes a String argument, converts it to a stream of bytes, and then writes it to disk.

Figure 18-8 shows the results of running this program. Notice that the writeBytes() method behaves somewhat differently than the write() method in that it actually waits until the buffer is completely full and then flushes its contents to the file. Figure 18-9 shows the contents of test.txt when the program completes execution.

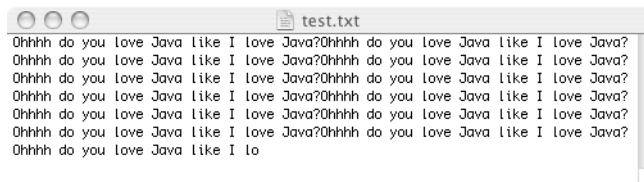


```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/dos_tester] swodog% java DOS_TesterApp
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 0
The length of the file Text.txt in bytes is: 512
[Rick-Millers-Computer:~/desktop/dos_tester] swodog%

```

Figure 18-8: Results of Running Example 18.5



```

test.txt
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?
Ohhhh do you love Java like I lo

```

Figure 18-9: Contents of test.txt After Running Example 18.5

ObjectOutputStream

The ObjectOutputStream class can be used in place of the DataOutputStream to perform sequential byte writes to a file. Additionally, you can use ObjectOutputStream to write primitive types and serialized objects to a file as well. To use an ObjectOutputStream object in your program you first create a FileOutputStream, then, optionally, a BufferedOutputStream, and finally the ObjectOutputStream. Objects written to disk must implement the Serializable or the Externalizable interface.

I will demonstrate the functionality of the ObjectOutputStream class by showing you how to write a collection of Person objects to a file. Example 18.6 gives the code for the simplified Person class.

```

1     import java.io.*;
2
3     public class Person implements Serializable {
4
5         private String _f_name;
6         private String _m_name;
7         private String _l_name;
8         private String _gender;
9

```

18.6 Person.java

```

10     public Person(String f_name, String m_name, String l_name, String gender){
11         _f_name = f_name;
12         _m_name = m_name;
13         _l_name = l_name;
14         _gender = gender;
15     }
16
17     public String getFirstName(){ return _f_name; }
18     public String getMiddleName(){ return _m_name; }
19     public String getLastName(){ return _l_name; }
20     public String getGender(){ return _gender; }
21     public String toString(){ return _f_name + " " + _m_name + " " + _l_name + " " + _gender; }
22
23 }

```

This simplified Person class provides fields for a Person's first name, middle name, last name, and gender, along with a set of related accessor methods. Example 18-7 gives the code for a program that creates a Java generic Vector<Person> and writes the entire vector to disk.

18.7 OOSTesterApp.java

```

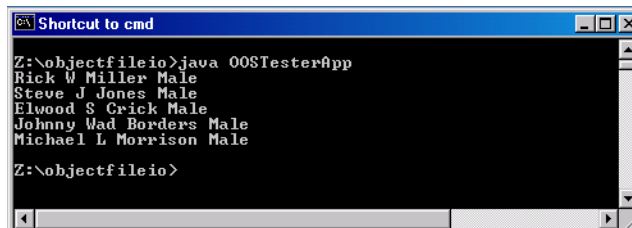
1     import java.io.*;
2     import java.util.*;
3
4     public class OOSTesterApp {
5         public static void main(String[] args){
6             Vector<Person> people_vector = new Vector<Person>();
7             people_vector.add(new Person("Rick", "W", "Miller", "Male"));
8             people_vector.add(new Person("Steve", "J", "Jones", "Male"));
9             people_vector.add(new Person("Elwood", "S", "Crick", "Male"));
10            people_vector.add(new Person("Johnny", "Wad", "Borders", "Male"));
11            people_vector.add(new Person("Michael", "L", "Morrison", "Male"));
12
13            for(int i = 0; i<5; i++){
14                System.out.println(people_vector.elementAt(i).toString());
15            }
16
17            FileOutputStream fos;
18            ObjectOutputStream oos;
19
20            try{
21                fos = new FileOutputStream("People.dat");
22                oos = new ObjectOutputStream(fos);
23                oos.writeObject(people_vector);
24                fos.close();
25            }catch(Exception ignored){}
26
27        } //end main
28    } //end class

```

Referring to example 18.7 — the Vector<Person> reference named people_vector is created on line 6. On lines 7 through 11, five Person objects are created and added to the people_vector. The for statement on line 13 simply iterates through the people_vector and calls the toString() method on each contained Person object.

On lines 17 and 18 the FileOutputStream and ObjectOutputStream references are declared. These are used in the try block beginning on line 20 to create the file and write the people_vector object to disk. Any exceptions that may be generated are ignored in this example.

Figure 18-10 shows the results of running example 18.7. Figure 18.11 shows how the contents of the People.dat file appear when viewed with a text editor.



```

Shortcut to cmd
Z:\objectfileio>java OOSTesterApp
Rick W Miller Male
Steve J Jones Male
Elwood S Crick Male
Johnny Wad Borders Male
Michael L Morrison Male
Z:\objectfileio>

```

Figure 18-10: Results of Running Example 18.7

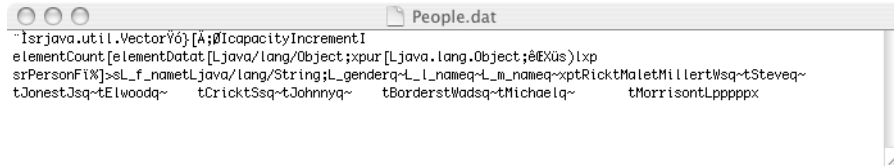


Figure 18-11: Contents of People.dat File Viewed with Text Editor

PRINTSTREAM

The `PrintStream` class is used to write to file textual representations of Java primitive types. You can also use the class to write bytes, or arrays of bytes, to a file as well. To use the `PrintStream` class you must first create a `FileOutputStream` object. The `PrintStream` class offers three overloaded constructors that let you specify the output text encoding scheme and whether or not to automatically flush the buffer each time a write or a print is performed.

Example 18.8 demonstrates the use of the `PrintStream` class. This example uses the simplified `Person` class listed in example 18.6.

18.8 *PrintStreamTestApp.java*

```

1      import java.io.*;
2
3      public class PrintStreamTestApp {
4          public static void main(String[] args){
5              try{
6                  PrintStream ps = new PrintStream(new FileOutputStream("Output.txt"));
7                  ps.println("This file created with a PrintStream object!"); // print String literal
8                  ps.println(343); // print integer literal
9                  ps.println(2.111); // print float literal
10                 ps.println(false); // print boolean literal
11                 ps.println(new Person("Rick", "W", "Miller", "Male")); // print Object String
12                 ps.close();
13             }catch(Exception ignored){}
14         } // end main
15     } // end class

```

Referring to example 18.8 — The `PrintStream` object is created inside the try block on line 6. Notice how the `FileOutputStream` object is created “on the fly” as an argument to the `PrintStream` constructor. You will see this shortcut technique used often in Java programming examples.

The `ps` reference is used to write the text forms of various objects to the `Output.txt` file. A `String` literal is written on line 7, followed by an integer literal on line 8, a double literal on line 9, and a boolean literal on line 10. A `Person` object is written on line 11. But what actually gets written to the file? Figure 18-12 shows the contents of the `Output.txt` file viewed with a text editor.

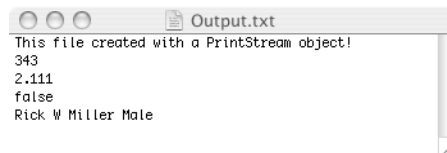


Figure 18-12: Contents of Output.txt File After Example 18.8 Executes

Notice how the `String` representation of the `Person` object is written to the file. This happened because the `Person` object supplied a `toString()` method, which overrides the `Object.toString()` method. If you want to save textual representations of your objects be sure to provide an overriding `toString()` method. (*Note: The `System.out` object is a `PrintStream` object.*)

INPUTSTREAMS

In this section I will demonstrate the use of the `InputStream` classes by reading and displaying the contents of the files created in the preceding sections with the `OutputStream` classes. The classes demonstrated in this section include the `FileInputStream`, `BufferedInputStream`, `DataInputStream`, and `ObjectInputStream`.

InputStream classes read input sequentially from the start of the file to the end of the file. You can read all the bytes in a file at one time or read various parts of the file sequentially. Your choice of InputStream class is dictated mostly by what type of data is stored in the file and how you want to read it. Technically, you can read any file type with these classes but the data contained within that file may or may not be interpreted properly. You generally need apriori knowledge of a file's content organization to properly read and interpret a file's data.

FileInputStream

The FileInputStream is a file-terminal class. In most situations where you want to read a byte stream file you need to create a FileInputStream object first, followed by either a BufferedInputStream, DataInputStream, or an ObjectInputStream object. Example 18.9 shows a FileInputStream object being used to read the Text.txt file created with the execution of example 18.3.

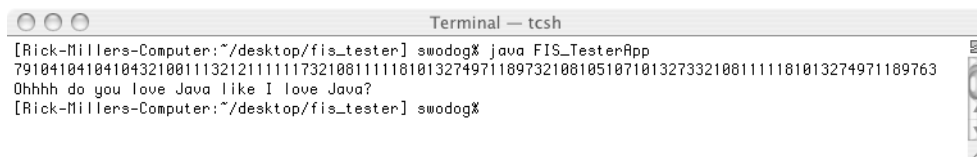
18.9 FIS_TesterApp.java

```

1      import java.io.*;
2
3      public class FIS_TesterApp {
4          public static void main(String[] args){
5              try{
6                  File file = new File("Text.txt");
7                  FileInputStream fis = new FileInputStream(file);
8                  byte[] b = new byte[(int)file.length()];
9                  fis.read(b);
10
11                 for(int i = 0; i<b.length; i++){
12                     System.out.print(b[i]);          // print each byte
13                 }
14
15                 System.out.println();
16
17                 for(int i = 0; i<b.length; i++){
18                     System.out.print((char)b[i]);    // cast to char then print
19                 }
20
21                 System.out.println();
22
23                 fis.close();
24             }catch(Exception ignored){ }
25         } // end main
26     } // end class

```

Referring to example 18.9 — The File and FileInputStream objects are created on lines 6 and 7. A byte array the length of the file is created on line 8. The entire file is then read into the byte array on line 9. The for statement on line 11 prints each element of the byte array to the console as a byte. The for statement on line 17 prints each byte in the array after it has been cast to a character. Figure 18-13 shows the results of running this program.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/fis_tester] swodog% java FIS_TesterApp
7910410410410432100111321211111173210011111810132749711897321001051071013273321001111181013274971189763
Ohhhh do you love Java like I love Java?
[Rick-Millers-Computer:~/desktop/fis_tester] swodog%

```

Figure 18-13: Results of Running Example 18.9

Can you correlate each character to its ASCII code?

BufferedInputStream

The BufferedInputStream class is used to provide file read buffering. You can set the size of its internal buffer via the constructor at the time of object creation. Its default buffer size is 512 bytes. Example 18.10 demonstrates the use of a BufferedInputStream object.

18.10 BIS_TesterApp.java

```

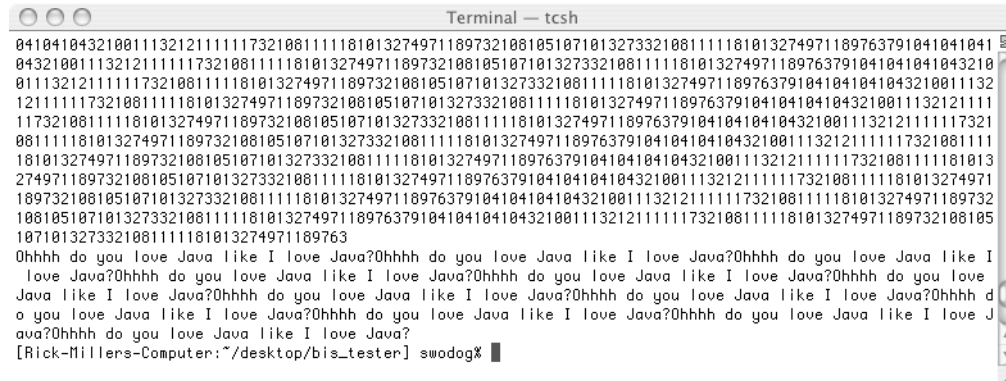
1      import java.io.*;
2
3      public class BIS_TesterApp {
4          public static void main(String[] args){
5              try{

```



```
6         File file = new File("Test.txt");
7         FileInputStream fis = new FileInputStream(file);
8         BufferedInputStream bis = new BufferedInputStream(fis);
9         byte[] b = new byte[(int)file.length()];
10        bis.read(b);
11
12        for(int i = 0; i<b.length; i++){
13            System.out.print(b[i]);           // print each byte
14        }
15
16        System.out.println();
17
18        for(int i = 0; i<b.length; i++){
19            System.out.print((char)b[i]);     // cast to char then print
20        }
21
22        System.out.println();
23
24        bis.close();
25    }catch(Exception ignored){ }
26    } // end main
27 } // end class
```

Referring to example 18.10 — the code for this example is similar to example 18.9 except now a `BufferedInputStream` object is created on line 8 and is used to conduct file input operations from that point on. The results of running this program are shown in figure 18-14. The version of the `Test.txt` file used in this example is the one created in example 18.4.



```
Terminal — tcsh
0410410432100111321211111173210811111810132749711897321081051071013273321081111181013274971189763791041041041
0432100111321211111173210811111810132749711897321081051071013273321081111181013274971189763791041041041043210
0111321211111173210811111810132749711897321081051071013273321081111181013274971189763791041041041043210011132
1211111173210811111810132749711897321081051071013273321081111181013274971189763791041041041043210011132121111
117321081111181013274971189732108105107101327332108111118101327497118976379104104104104321001113212111117321
081111181013274971189732108105107101327332108111118101327497118976379104104104104321001113212111117321081111
181013274971189732108105107101327332108111118101327497118976379104104104104321001113212111117321081111181013
2749711897321081051071013273321081111181013274971189763791041041041043210011132121111117321081111181013274971
1897321081051071013273321081111181013274971189763791041041041043210011132121111117321081111181013274971189732
1081051071013273321081111181013274971189763791041041041043210011132121111117321081111181013274971189732108105
1071013273321081111181013274971189763
Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I
 love Java?Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?Ohhhh do you love
Java like I love Java?Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?Ohhhh do you love
o you love Java like I love Java?Ohhhh do you love Java like I love Java?Ohhhh do you love Java like I love Java?Ohhhh d
ava?Ohhhh do you love Java like I love Java?
[Rick-Millers-Computer:~/desktop/bis_tester] swdogX
```

Figure 18-14: Results of Running Example 18.10.

DATAINPUTSTREAM

The `DataInputStream` class provides numerous interface methods and is used to read bytes, arrays of bytes, primitive types, and text from a file. Example 18.11 shows how to use a `DataInputStream` object to read a series of Strings from a file. The file used as a source of input in this example is that created with the `PrintStream` class in example 18.8.

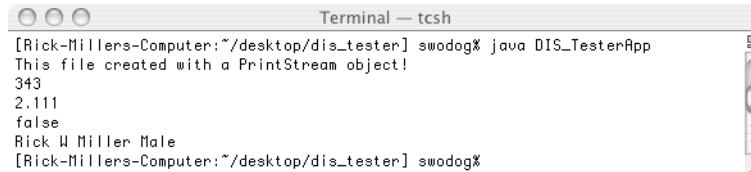
18.11 *DIS_TesterApp.java*

```
1     import java.io.*;
2
3     public class DIS_TesterApp {
4         public static void main(String[] args){
5             try{
6                 File file = new File("Output.txt");
7                 FileInputStream fis = new FileInputStream(file);
8                 DataInputStream dis = new DataInputStream(fis);
9
10                String input = null;
11                while((input = dis.readLine()) != null){ //readLine() method deprecated
12                    System.out.println(input);
13                }
14                dis.close();
15            }catch(Exception ignored){}
16        } // end main
17    } //end class
```

Referring to example 18.11 — the file object is created on line 6. It is used to create the `FileInputStream` object on line 7, which is used in turn to create the `DataInputStream` object on line 8. The while loop on line 11 reads a line of strings at a time until it encounters a null string.

The `readLine()` method used on line 11 is deprecated and its use will result in a compiler warning. Although this program works today, there is no guarantee it will work in future versions of Java. If you want to read text from a file you are advised to use a `Reader` class. Readers are covered later in this chapter.

Figure 18-15 shows the results of running this program.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/dis_tester] swodog% java DIS_TesterApp
This file created with a PrintStream object!
343
2.111
false
Rick W Miller Male
[Rick-Millers-Computer:~/desktop/dis_tester] swodog%
```

Figure 18-15: Results of Running Example 18.11

ObjectInputStream

The `ObjectInputStream` class is used to read serialized objects from a file. With it you can read individual objects or collections of objects that have been serialized as part of an array or collection object. Example 18.12 gives the code for a class that reads the `Vector<Person>` object written to disk in example 18.6.

18.12 *OISTesterApp.java*

```
1      import java.io.*;
2      import java.util.*;
3
4      public class OISTesterApp {
5          public static void main(String[] args){
6              Vector<Person> people_vector = null;
7
8              try{
9                  FileInputStream  fis = new FileInputStream("People.dat");
10                 ObjectInputStream ois = new ObjectInputStream(fis);
11                 people_vector = (Vector<Person>) ois.readObject();
12
13                 for(int i = 0; i<5; i++){
14                     System.out.println(people_vector.elementAt(i).toString());
15                 }
16
17                 ois.close();
18             }catch(Exception e){System.out.println(e.getMessage()); }
19         }
20     }
```

Referring to example 18.12 — A `Vector<Person>` reference named `people_vector` is declared on line 6. The `FileInputStream` and `ObjectInputStream` objects are declared and created on lines 9 and 10. On line 11 the `ois` reference is used to read the `People.dat` file. The deserialized object must be cast to the proper type, which is, in this case, `Vector<Person>`. When this source file is compiled under Java version 5 it produces the warning shown in figure 18-16.

The reason for the warning? Deserialized objects are always problematic in that you cannot guarantee fully the type of object you are deserializing. Java version 5 is a little more vocal about the possible problems with deserializing objects. Figure 18-17 shows the results of running example 18.12.

Quick Review

Use `OutputStreams` and `InputStreams` to write and read byte-oriented stream data to and from a file to include single bytes, arrays of bytes, Java primitive types, serialized objects, and text.

```

Z:\objectfileio>javac OISTesterApp.java
Note: OISTesterApp.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

Z:\objectfileio>javac -Xlint:unchecked OISTesterApp.java
OISTesterApp.java:11: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: java.util.Vector<Person>
    people_vector = <Vector<Person>> ois.readObject();
1 warning
Z:\objectfileio>_

```

Figure 18-16: Warning Produced When Compiling Example 18.12

```

Z:\objectfileio>java OISTesterApp
Rick W Miller Male
Steve J Jones Male
Elwood S Crick Male
Johnny Wad Borders Male
Michael L Morrison Male
Z:\objectfileio>

```

Figure 18-17: Results of Running Example 18.12

SEQUENTIAL CHARACTER STREAM FILE I/O USING READERS & WRITERS

The Reader and Writer classes support character-based file I/O more robustly than do their `InputStream` and `OutputStream` counterparts. The primary reason for this is their support for specifying the character set encoding scheme used to translate bytes or characters from the local file encoding to that required for the application. If you have a need to perform character-based file I/O then look to the Reader and Writer classes for your I/O solution.

As you can tell from looking at the inheritance hierarchy shown in figure 18-1 there are notably fewer Reader and Writer classes than there are `InputStream` and `OutputStream` classes. This is because Readers and Writers focus on character input and output. However, as you see the Readers and Writers put to use in this section, you will notice that for each Reader and Writer there is a corresponding `InputStream` and `OutputStream` class that performs a somewhat similar function. Chronologically during the evolution of the Java Platform API the `InputStream` and `OutputStream` classes appeared first followed by the Readers and Writers. It is their support for internationalization that makes Readers and Writers the preferred choice for providing character-based file I/O functionality.

WRITERS

In this section I will demonstrate the use of the `FileWriter`, `BufferedWriter`, and `PrintWriter` class. If you understood the material presented earlier about the `OutputStream` classes you can make a deadly accurate guess as to the intended purpose of a `FileWriter`, `BufferedWriter`, and `PrintWriter`. You'll use a `FileWriter` to create a file to write characters to and a `BufferedWriter` to improve output efficiency by buffering the writes. The `PrintWriter` works a lot like the `PrintStream` class and allows you to write textual representations of Java primitive types and objects.

FileWriter

The `FileWriter` class has only five methods and they are all constructors. Its functionality derives from its base class `OutputStreamWriter`. When you create an object of type `FileWriter` it automatically creates an internal `FileOutputStream` object that it uses to write bytes to a file.

The `FileWriter` class is an exception to the flexible internationalization capability provided by `Writer` classes in general. When using a `FileWriter` class you are limited to writing characters to the default local encoding. If you need to specify an encoding other than the default you must use an `OutputStreamWriter` in conjunction with a `FileOutputStream` class.

Example 18.13 shows the `FileWriter` class in action.

18.13 *FWTesterApp.java*

```

1      import java.io.*;
2
3      public class FWTesterApp {
4          public static void main(String[] args){
5              try{
6                  FileWriter fw = new FileWriter("test.txt");
7                  fw.write("This is a test of the FileWriter class.\n");
8                  fw.write("It inherits its functionality from OutputStreamWriter.\n");
9                  fw.write("You can overwrite the contents of a file...\n");
10                 fw.write("...or you can append text to the end of a file.");
11                 fw.close();
12             }catch(Exception ignored){ }
13         } // end main
14     } // end class

```

Referring to example 18.13 — the `FileWriter` class stands pretty much on its own. To use it you simply create an object of type `FileWriter`, specifying the name of the file in the form of either a `String` or `File` reference, and you're ready to write text to a file. The `write()` methods are provided by the `FileWriter`'s superclass `OutputStreamWriter`. Notice how, if you want to write lines of text, you have to manually insert the newline character `'\n'`. Figure 18-18 shows the contents of the `test.txt` file as viewed with a text editor after running this program.

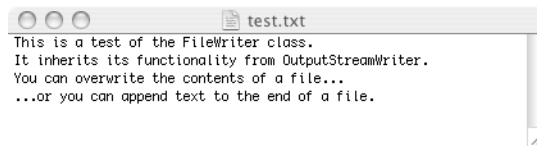


Figure 18-18: Contents of `test.txt` File After Example 18.13 Executes

BufferedWriter

The `BufferedWriter` class provides output buffering for other `Writer` objects. It is used mostly in conjunction with a `FileWriter` or `OutputStreamWriter` object. Example 18.14 shows the `BufferedWriter` class in action.

18.14 *BWTesterApp.java*

```

1      import java.io.*;
2
3      public class BWTesterApp {
4          public static void main(String[] args){
5              try{
6                  FileWriter fw = new FileWriter("test.txt");
7                  BufferedWriter bw = new BufferedWriter(fw);
8                  bw.write("This is a test of the BufferedWriter class.");
9                  bw.newLine();
10                 bw.write("It improves output efficiency by buffering individual disk writes.");
11                 bw.newLine();
12                 bw.write("You can specify the internal buffer size at the time of object creation.");
13                 bw.newLine();
14                 bw.write("...and it provides a newLine() method!");
15                 bw.flush();
16                 bw.close();
17             }catch(Exception ignored){ }
18         } // end main
19     } // end class

```

Referring to example 18.14 — the `FileWriter` object is created first and is used to create the `BufferedWriter` object. You can specify the size of its internal buffer via a constructor although in this example the default buffer size is utilized. The `BufferedWriter` object also provides a `newLine()` method. Figure 18-19 shows the contents of the `test.txt` file as viewed with a text editor after this program executes.

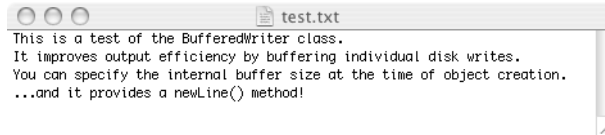


Figure 18-19: Contents of test.txt After Example 18.14 Executes

OUTPUTSTREAMWRITER

The `OutputStreamWriter` provides character encoding flexibility. To use this class you must first create an `OutputStream` object. In this example a `FileOutputStream` will be used since we are writing to files.

If you need to provide output buffering you can use an `OutputStreamWriter` in conjunction with a `BufferedWriter`. Example 18.15 shows the `OutputStreamWriter` in action.

18.15 *OSWTesterApp.java*

```

1      import java.io.*;
2
3      public class OSWTesterApp {
4          public static void main(String[] args){
5              try{
6                  FileOutputStream fos = new FileOutputStream("test.txt");
7                  OutputStreamWriter osw = new OutputStreamWriter(fos);
8                  BufferedWriter bw = new BufferedWriter(osw);
9                  bw.write("Ohhhh do you love Java like I love Java?");
10                 bw.newLine();
11                 bw.write("The OutputStreamWriter needs a FileOutputStream object...");
12                 bw.newLine();
13                 bw.write("...and can be used with a BufferedWriter to improve output efficiency.");
14                 bw.newLine();
15                 bw.write("...and it's fun to use!");
16                 bw.flush();
17                 bw.close();
18             }catch(Exception ignored){ }
19         } // end main
20     } // end class

```

Referring to example 18.15 — the `OutputStreamWriter` needs to work with a `FileOutputStream` object in order to do its job. It can also be combined, as it is in this example, with a `BufferedWriter` to improve output efficiency. Figure 18-20 shows the contents of the test.txt file as viewed with a text editor.

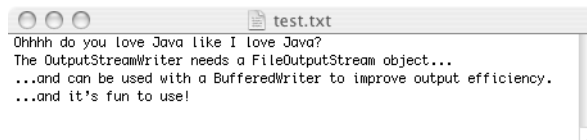


Figure 18-20: Contents of test.txt File After Example 18.15 Executes

PRINTWRITER

The `PrintWriter` class provides functionality similar to that of the `PrintStream` class in that it is used to write textual representations of Java primitive types and objects, as well as `Strings` and `chars`, to disk or other output destination. The `PrintWriter` must be used in conjunction with either an `OutputStream` object such as a `FileOutputStream` or a `Writer` object such as an `OutputStreamWriter`. When used with an `OutputStreamWriter` you can specify the desired character encoding. Example 18.16 shows the `PrintWriter` class in action.

18.16 *PWTesterApp.java*

```

1      import java.io.*;
2
3      public class PWTesterApp {
4          public static void main(String[] args){
5              try{
6                  FileOutputStream fos = new FileOutputStream("test.txt");
7                  OutputStreamWriter osw = new OutputStreamWriter(fos);
8                  BufferedWriter bw = new BufferedWriter(osw);
9                  PrintWriter pw = new PrintWriter(bw);

```

```

10         pw.println(123); // print integer literal
11         pw.println(true); // print boolean literal
12         pw.println(1.234); // print double literal
13         pw.println("Ohhhh I love Java!!"); // print String literal
14         pw.println(new Person("Rick", "W", "Miller", "Male")); // print Object String representation
15         pw.flush();
16         pw.close();
17     }catch(Exception ignored){ }
18 } // end main
19 } // end class

```

Referring to example 18.16 — The `PrintWriter` class is being used with a `BufferedWriter` to write character representations of Java primitive types and objects. In this example, the `Person` class is used to create a `Person` object on line 14. Figure 18-21 shows the contents of the `test.txt` file as viewed with a text editor after example 18.16 executes.



Figure 18-21: Contents of `test.txt` File After Example 18.16 Executes

READERS

Reader classes are used to read characters from files or other character stream sources. Readers allow you to specify the character encoding used to translate the bytes read from the data source.

FileReader

The `FileReader` class reads bytes from a file and translates them according to the default local encoding. If you need more control over the encoding scheme use the `InputStreamReader` class directly. Example 18.17 shows the `FileReader` class in action.

18.17 `FRTesterApp.java`

```


1     import java.io.*;
2
3     public class FRTesterApp {
4         public static void main(String[] args){
5             try{
6                 File file = new File("test.txt");
7                 FileReader fr = new FileReader(file);
8                 char[] char_buffer = new char[(int)file.length()];
9                 fr.read(char_buffer);
10
11                 for(int i = 0; i<char_buffer.length; i++){
12                     System.out.print(char_buffer[i]);
13                 }
14                 fr.close();
15             }catch(Exception ignored){ }
16         } // end main
17     } // end class

```

Referring to example 18.17 — A `File` object is created on line 6 and is used to create the `FileReader` object on line 7. The `File` object is then used to get the length of the `test.txt` file and use that value (*cast to an integer*) to create a character array named `char_buffer`. The `char_buffer` reference is then supplied as an argument to the `read()` method on line 9 where the entire contents of the `test.txt` file is read into the `char_buffer` array. The `for` statement on line 11 iterates over the `char_buffer` array and prints each character to the console. Figure 18-22 shows the results of running this program.

InputStreamReader

The `InputStreamReader` class can be used when you need more control over the encoding scheme used to translate the bytes read from a file into characters. The `InputStreamReader` requires the services of an `InputStream` object



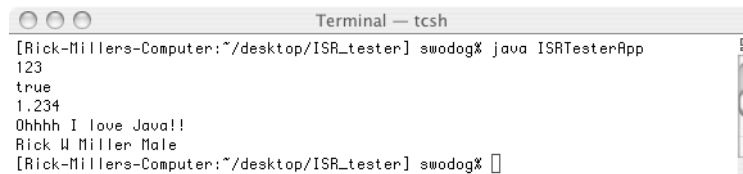
```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/fr_tester] swodog% java FRTesterApp
123
true
1.234
Ohhhh I love Java!!
Rick W Miller Male
[Rick-Millers-Computer:~/desktop/fr_tester] swodog% 
```

Figure 18-22: Results of Running Example 18.17

to help it do its job. Example 18.18 uses an `InputStreamReader` in conjunction with a `FileInputStream`. Figure 18-23 shows the results of running this program.

18.18 ISRTTesterApp.java

```
1      import java.io.*;
2
3      public class ISRTTesterApp {
4          public static void main(String[] args){
5              try{
6                  File file = new File("test.txt");
7                  FileInputStream fis = new FileInputStream(file);
8                  InputStreamReader isr = new InputStreamReader(fis);
9                  char[] char_buffer = new char[(int)file.length()];
10                 isr.read(char_buffer);
11
12                 for(int i = 0; i<char_buffer.length; i++){
13                     System.out.print(char_buffer[i]);
14                 }
15
16                 isr.close();
17             }catch(Exception ignored){ }
18         } // end main
19     } // end class
```



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/ISR_tester] swodog% java ISRTTesterApp
123
true
1.234
Ohhhh I love Java!!
Rick W Miller Male
[Rick-Millers-Computer:~/desktop/ISR_tester] swodog% 
```

Figure 18-23: Results of Running Example 18.18

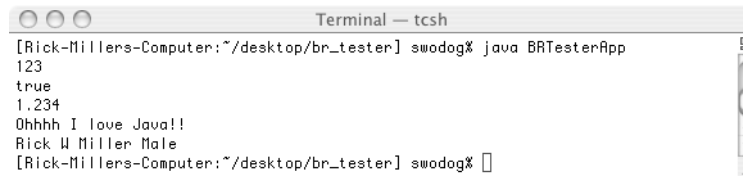
BufferedReader

The `InputStreamReader` can be used in conjunction with a `BufferedReader` to improve input efficiency. The size of the `BufferedReader`'s internal buffer can be specified at the time of instantiation. Example 18.19 shows a `BufferedReader` in action.

18.19 BRTesterApp.java

```
1      import java.io.*;
2
3      public class BRTesterApp {
4          public static void main(String[] args){
5              try{
6                  File file = new File("test.txt");
7                  FileInputStream fis = new FileInputStream(file);
8                  InputStreamReader isr = new InputStreamReader(fis);
9                  BufferedReader br = new BufferedReader(isr);
10                 char[] char_buffer = new char[(int)file.length()];
11                 br.read(char_buffer);
12
13                 for(int i = 0; i<char_buffer.length; i++){
14                     System.out.print(char_buffer[i]);
15                 }
16
17                 br.close();
18             }catch(Exception ignored){ }
19         } // end main
20     } // end class
```

Referring to example 18.19 — the `BufferedReader` object is created on line 9 using the `InputStreamReader` reference as an argument to its constructor. Figure 18-24 shows the results of running this program.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/br_tester] swodog% java BRTesterApp
123
true
1.234
Ohhhh I love Java!!
Rick W Miller Male
[Rick-Millers-Computer:~/desktop/br_tester] swodog% █
```

Figure 18-24: Results of Running Example 18.19

Quick Review

The `Writer` and `Reader` classes enable you to write and read character-oriented stream data to and from a file. If you need more control over the character encoding used in your program use the `Reader` and `Writer` classes for your file I/O operations.

A PRACTICAL FILE I/O EXAMPLE: THE `JAVA.UTIL.PROPERTIES` CLASS

There are many classes in the Java Platform API that depend on the `InputStream` and `OutputStream` family of classes in some way or another. An example of one such class is the `Properties` class.

The `Properties` class provides a convenient way for you to store application properties in two types of plain text file formats: (*property_name/value pairs*) or xml files (`<properties><entry> tags`). Example 18.20 gives the code for a class named `AppProperties` and example 18.21 shows the `AppProperties` class in action.

18.20 *AppProperties.java*

```
1      import java.util.*;
2      import java.io.*;
3
4      public class AppProperties extends Properties {
5          /*
6           * Declare default property names and their values
7           */
8          public static final String PROPERTY_A_NAME = "PROPERTY_A_NAME";
9          public static final String PROPERTY_B_NAME = "PROPERTY_B_NAME";
10         public static final String PROPERTY_C_NAME = "PROPERTY_C_NAME";
11         public static final String DEFAULT_PROPERTIES_FILENAME = "DEFAULT_PROPERTIES_FILENAME";
12         public static final String PROPERTIES_FILE_HEADER = "PROPERTIES_FILE_HEADER";
13
14         private static final String PROPERTY_A_VALUE = "Property A's Value";
15         private static final String PROPERTY_B_VALUE = "Property B's Value";
16         private static final String PROPERTY_C_VALUE = "Property C's Value";
17         private static final String DEFAULT_PROPERTIES_FILENAME_VALUE = "app_prop.xml";
18         private static final String PROPERTIES_FILE_HEADER_VALUE = "Application Properties File";
19
20         /**
21          * Constructor
22          */
23
24         public AppProperties(){
25             System.out.println("Attempting to read properties file...");
26             FileInputStream fis = null;
27             FileOutputStream fos = null;
28             try{
29                 fis = new FileInputStream(DEFAULT_PROPERTIES_FILENAME_VALUE);
30                 this.loadFromXML(fis);
31             }catch(Exception e1){
32                 System.out.println("AppProperties: Problem loading properties file.");
33                 System.out.println(e1.toString());
34                 System.out.println("Creating new default properties file.");
35                 this.setProperty(PROPERTY_A_NAME, PROPERTY_A_VALUE);
36                 this.setProperty(PROPERTY_B_NAME, PROPERTY_B_VALUE);
37                 this.setProperty(PROPERTY_C_NAME, PROPERTY_C_VALUE);
38                 this.setProperty(DEFAULT_PROPERTIES_FILENAME, DEFAULT_PROPERTIES_FILENAME_VALUE);
39                 this.setProperty(PROPERTIES_FILE_HEADER, PROPERTIES_FILE_HEADER_VALUE);
```



```

40         try{
41             fos = new FileOutputStream(DEFAULT_PROPERTIES_FILENAME_VALUE);
42             this.storeToXML(fos, PROPERTIES_FILE_HEADER_VALUE);
43         }catch(Exception e2){
44             System.out.println("AppProperties: Problem creating default properties file.");
45             System.out.println(e2.toString());
46         }finally{
47             if(fis != null){
48                 try{
49                     fis.close();
50                 }catch(Exception ignored){ }
51             }
52
53             if(fos != null){
54                 try{
55                     fos.close();
56                 }catch(Exception ignored){ }
57             }
58
59             }// end finally
60         }// end primary try/catch block
61     }// end constructor
62 }// end AppProperties class definition

```

18.21 PropertiesTesterApp.java

```

1     public class PropertiesTesterApp {
2         public static void main(String[] args){
3             AppProperties properties = new AppProperties();
4             System.out.println(properties.getProperty(properties.PROPERTY_A_NAME));
5         }
6     }

```

Referring to example 18-20 — the AppProperties class extends the java.util.Properties class. The Properties class stores properties in a HashMap. A property consists of a property name (*property key*) and a property *value*. These name/value pairs are inserted into a Properties object's HashMap for later retrieval and use in a program. As I mentioned above, a Properties object can persist its collection of property name/value pairs to disk in two plain text formats: 1) name/value pairs or 2) xml tagged entries. The AppProperties class defines several default application properties: three bogus properties for example purposes and two real properties. The two real properties include the default name of the application's property file and a default properties file header string.

When an AppProperties object is created, an attempt is made to open and read the default properties file. If the open is successful the properties file is read and the AppProperties object's HashMap is initialized. (*This HashMap initialization takes place automatically.*) If, however, the file does not exist or cannot be opened, the AppProperties object initializes its HashMap to a set of default properties and then attempts to create the properties file and save those properties to the file. This example creates an xml tagged properties file by using the storeToXML() method.

Referring to example 18.21 — the PropertiesTesterApp simply creates an instance of an AppProperties object. Once the object is created the getProperty() method is called to retrieve the value of the property named PROPERTY_A_NAME.

Figure 18-25 shows the PropertiesTesterApp program being executed for the first time. (*i.e., the app_prop.xml file does not initially exist*) Figure 18-26 shows the contents of the app_prop.xml file after the program executes.

```

Shortcut to cmd
Z:\propertiestest>java PropertiesTesterApp
Attempting to read properties file...
AppProperties: Problem loading properties file.
java.io.FileNotFoundException: app_prop.xml (The system cannot find the file specified)
Creating new default properties file.
Property A's Value
Z:\propertiestest>_

```

Figure 18-25: Initial Execution of PropertiesTesterApp (Example 18.21)

```

Z:\proprietestest>
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Application Properties File</comment>
<entry key="DEFAULT_PROPERTIES_FILENAME">app_prop.xml</entry>
<entry key="PROPERTY_C_NAME">Property C's Value</entry>
<entry key="PROPERTY_B_NAME">Property B's Value</entry>
<entry key="PROPERTY_A_NAME">Property A's Value</entry>
<entry key="PROPERTIES_FILE_HEADER">Application Properties File</entry>
</properties>

```

Figure 18-26: Contents of app_prop.xml After Example 18.21 Executes

Quick Review

The Properties class is an example of a Java API class that depends on the services of the InputStream and OutputStream classes to persist property data to a file.

Random Access File I/O

The RandomAccessFile can be used to read and write bytes, characters, and primitive types — but not objects. It provides a seek() method that allows you to go to any point within the file, forward or backward.

In this section I will present a comprehensive example of the use of the RandomAccessFile class. The task at hand will be to create a Java adapter to a legacy application flat-file data store. The complete project specification is given in figure 18-27.

As you can see from the project specification this is a fairly complex project. A Java adapter must be written to access a legacy system's fixed-length record data files. You are given three artifacts to assist in your effort: 1) The data file schema definition is given in the project, 2) the example legacy data file is located on the accompanying disk or can be downloaded from www.pulpfreepress.com, and 3) a Java interface that provides declarations for the methods that must be implemented in the legacy data file adapter class. The interface code is listed in example 18.22

18.22 LegacyDatafileInterface.java

```

1      public interface LegacyDatafileInterface {
2
3          /**
4           * Read the record indicated by the rec_no and return a string array
5           * were each element contains a field value.
6           */
7          public String[] readRecord(long rec_no) throws RecordNotFoundException;
8
9
10         /**
11          * Update a record's fields. The record must be locked with the lockRecord()
12          * method and the lock_token must be valid. The value for field n appears in
13          * element record[n].
14          */
15         public void updateRecord(long rec_no, String[] record, long lock_token) throws
16         RecordNotFoundException, SecurityException;
17
18
19         /**
20          * Marks a record for deletion by setting the deleted field to 1. The lock_token
21          * must be valid otherwise a SecurityException is thrown.
22          */
23         public void deleteRecord(long rec_no, long lock_token) throws
24         RecordNotFoundException, SecurityException;
25
26
27         /**
28          * Creates a new datafile record and returns the record number.
29          */
30         public long createRecord(String[] record) throws FailedRecordCreationException;
31
32
33         /**
34          * Locks a record for updates and deletes and returns an integer

```

Project Specification

Objectives:

- Demonstrate your ability to conduct file I/O operations with the `RandomAccessFile` class
- Demonstrate your ability to implement a non-trivial interface
- Demonstrate your ability to translate low-level exceptions into higher-level, user-defined, application-specific exception abstractions
- Demonstrate your ability to coordinate file I/O operations via object synchronization

Tasks:

- You are a junior programmer working in the IT department of a retail bookstore. The CEO wants to begin migrating legacy systems to the web using Java technology. A first step in this initiative is to create Java adapters to existing legacy data stores. Given an interface definition, example legacy data file, and legacy data file schema definition, write a Java class that serves as an adapter object to a legacy data file.

Given:

- Java interface file specifying adapter operations
- Legacy data file schema definition
- Example legacy data file

Legacy Data File Schema Definition:

The legacy data file contains three sections:

- 1) The file identification section is a two-byte value that identifies the file as a data file.
- 2) The schema description section immediately follows the first section and contains the field text name and two-byte field length for each field in the data section.
- 3) The data section contains fixed-field-length record data elements arranged according to the following schema: (length is in bytes)

Field Name	Length	Description
deleted	1	numeric - 0 if valid, 1 if deleted
title	50	text - book title
author	50	text - author full name
pub_code	4	numeric - publisher code
ISBN	13	text - International Standard Book Number
price	8	text - retail price in following format: \$nnnn.nn
qoh	4	numeric - quantity on hand

Figure 18-27: Legacy Data File Adapter Project Specification

```

35     * representing a lock token.
36     */
37     public long lockRecord(long rec_no) throws RecordNotFoundException;
38
39
40     /**
41     * Unlocks a previously locked record. The lock_token must be valid or a
42     * SecurityException is thrown.
43     */
44     public void unlockRecord(long rec_no, long lock_token) throws SecurityException;
45
46
47     /**
48     * Searches the records in the datafile for records that match the String
49     * values of search_criteria. search_criteria[n] contains the search value
50     * applied against field n.
51     */
52     public long[] searchRecords(String[] search_criteria);
53
54 } //end interface definition

```

Referring to example 18.22 — the comments preceding each method declaration explain the functionality each method is required to provide. Most of the methods declare that they may throw exceptions that do not currently exist

in the Java API. As part of the project you will have to create each of these exception classes. Figure 18-28 shows the contents of the example legacy data file as viewed with a text editor.

```

zdeletedtitle2author2pub_codeISBN
priceqohC++ For Artists Rick Miller Rick Miller
1-932504-02-8$59.95 PC++ For Artists Rick Miller
1-932504-02-8$59.95 PJava For Artists Rick Miller
1-932504-04-X$69.95 dC++ For Artists Rick Miller
1-932504-02-8$59.95 PC++ For Artists Rick Miller
1-932504-02-8$59.95 PC++ For Artists Rick Miller
1-932504-02-8$59.95 PC++ For Artists Rick Miller
1-932504-02-8$59.95 PC++ For Artists Rick Miller
1-932504-02-8$59.95 PC++ For Artists Rick Miller
1-932504-02-8$59.95 PC++ For Artists Rick Miller
1-932504-02-8$59.95 PC++ For Artists Rick Miller
1-932504-02-8$59.95 PC++ For Artists Rick Miller

```

Figure 18-28: Contents of books.dat Example Legacy Datafile Viewed with Text Editor

TOWARDS AN APPROACH TO THE ADAPTER PROJECT

Given the project specification and the three supporting artifacts you may be wondering where to begin. Using the guidance offered by the project-approach strategy in chapter 1, I recommend devoting some time to studying the schema definition and comparing it with what you see in the example data file. You will note that although some of the text appears to read OK, there are a few characters here and there that seem out of place. For instance, you can make out the header information but the header appears to start with a letter 'z'. Studying the schema definition closely you note that the data file begins with a two-byte file identifier number. But what's the value of this number?

START SMALL AND TAKE BABY STEPS

One way to find out is to write a short program that reads the first two bytes of the file and converts it to a number. The `RandomAccessFile` class implements the `DataInput` and `DataOutput` interfaces. The `DataInput` interface has a method named `readShort()`. A Java short is a two-byte value. The `readShort()` method would be an excellent method to use to read the first two bytes of the file in an effort to determine their value.

The next phase of your discovery would be to try and read the rest of the file, or at least at first try and read the complete header and one complete record using the schema definition as a guide. You may find that a more detailed analysis of the header and record lengths are in order. Figure 18-29 shows a simple analysis performed with a spreadsheet.

	A	B	C	D	E	F	G
2	Header	Section 1	Magic Cookie	2			
3							
4		Section 2	deleted	7			
5			field length	2			
6			title	5			
7			field length	2			
8			author	6			
9			field length	2			
10			pub_code	8			
11			field length	2			
12			ISBN	4			
13			field length	2			
14			price	5			
15			field length	2			
16			qoh	3			
17			field length	2			
18							
19			Total Header Length	54			
20							
21							
22	Data		Field Name	Length in Bytes		Offset Start	Offset End
23			deleted	1		0	1
24			title	50		1	51
25			author	50		51	101
26			pub_code	4		101	105
27			ISBN	13		105	118
28			price	8		118	126
29			qoh	4		126	130
30							
31			Total Record Length	130			

Figure 18-29: Header and Record Length Analysis

Referring to figure 18-29 — the simple analysis reveals that the length of the header section of the legacy data file is 54 bytes long and each record is 130 bytes long. These figures, as well as the individual field lengths, will come in handy when the adapter is written.

Armed with some knowledge about the structure of the legacy data file and having gained some experience writing a small test program that reads all or portions of the file you can begin to create the adapter class incrementally. A good method to start with is the `readRecord()` method specified in the `LegacyDatafileInterface`.

OTHER PROJECT CONSIDERATIONS

This section briefly discusses additional issues which must be considered during the project implementation phase. These considerations include 1) record locking during updates and deletes, and 2) translating low-level I/O exceptions into higher level exceptions as specified in the interface.

Locking A Record For Updates And Deletes

The `LegacyDatafileInterface` specifies that a record must be locked when it is being updated or deleted. The locking is done via a lock token, which is nothing more than a primitive-type long value. How might the locking mechanism be implemented? How is the `lock_token` generated?

To implement the locking mechanism you must thoroughly understand threads and thread synchronization. (These topics are covered in detail in chapter 16.) Any object can be used as a synchronization point by using the `synchronized` keyword. The adapter must ensure that if one thread attempts to update or delete a record (by calling the `updateRecord()` or `deleteRecord()` methods) it cannot do so while another thread is in the process of calling either of those methods.

You can adopt several strategies as a means to an ends here. You can 1) apply the `synchronized` keyword to the entire method in question (`updateRecord()` and `deleteRecord()`) or 2) synchronize only the critical section of code within each method. Within the synchronized block you implement logic to check for a particular condition. If the condition holds you can proceed with whatever it is you need to do. If the condition does not hold, you will have to wait until it does. You do this by calling the `wait()` method on the object you have selected to synchronize on. The `wait()` method adds the current thread to a list of threads waiting to get a lock on that object.

Conversely, when a thread has obtained a lock on an object and it concludes its business, it must release its lock and notify the object upon which it is synchronized that it can wake up one of the other waiting threads. This is done by calling the `notify()` or `notifyAll()` method on the synchronized object.

Once the thread synchronization scheme is determined the `lock_token` can be generated with the `java.util.Random` class.

Translating Low-Level Exceptions Into Higher-Level Exception Abstractions

The `java.io` package defines several low-level exceptions that can occur when conducting file I/O operations. These exceptions must be handled in the adapter, however, the `LegacyDatafileInterface` specifies that several higher-level exceptions may be thrown when its methods are called.

To create custom exceptions you extend the `Exception` class and add any customized behavior required. (Exceptions are discussed in detail in chapter 15.) In your adapter code, you will catch and handle the low-level exceptions when they occur, and then repackage the exception within the context of a custom exception, and then throw the custom exception. Any objects utilizing the services of the adapter class must handle your custom exceptions, not the low-level I/O exceptions.

WHERE TO GO FROM HERE

The previous sections attempted to address some of the development issues you will typically encounter when attempting this type of project. The purpose of the project is to demonstrate the use of the `RandomAccessFile` class in the context of a non-trivial example. I hope also that I have sufficiently illustrated the reality that rarely can one class perform its job without the help of many other classes.

The next section gives the code for the completed project. Keep in mind that the examples listed here represent one particular approach and solution to the problem. As an exercise you will be invited to attempt a solution on your own terms using the knowledge gained here as a guide.

Explore and study the code. Compile the code and observe its operation. Experiment — make changes to areas you feel can use improvement.

COMPLETE RANDOMACCESSFILE LEGACY DATAFILE ADAPTER SOURCE CODE LISTING

This section gives the complete listing for the code that satisfies the requirements of the legacy data file adapter project using the RandomAccessFile class. The LegacyDatafileInterface listing given in example 18.22 is not repeated here.

18.23 DataFileAdapter.java

```

1      import java.io.*;
2      import java.util.*;
3
4
5      public class DataFileAdapter implements LegacyDatafileInterface {
6
7          /**
8           * Class Constants
9           */
10
11         private static final short FILE_IDENTIFIER = 378;
12         private static final int HEADER_LENGTH = 54;
13         private static final int RECORDS_START = 54;
14         private static final int RECORD_LENGTH = 130;
15         private static final int FIELD_COUNT = 7;
16
17         private static final short DELETED_FIELD_LENGTH = 1;
18         private static final short TITLE_FIELD_LENGTH = 50;
19         private static final short AUTHOR_FIELD_LENGTH = 50;
20         private static final short PUB_CODE_FIELD_LENGTH = 4;
21         private static final short ISBN_FIELD_LENGTH = 13;
22         private static final short PRICE_FIELD_LENGTH = 8;
23         private static final short QOH_FIELD_LENGTH = 4;
24
25         private static final String DELETED_STRING = "deleted";
26         private static final String TITLE_STRING = "title";
27         private static final String AUTHOR_STRING = "author";
28         private static final String PUB_CODE_STRING = "pub_code";
29         private static final String ISBN_STRING = "ISBN";
30         private static final String PRICE_STRING = "price";
31         private static final String QOH_STRING = "qoh";
32
33         private static final int TITLE_FIELD = 0;
34         private static final int AUTHOR_FIELD = 1;
35         private static final int PUB_CODE_FIELD = 2;
36         private static final int ISBN_FIELD = 3;
37         private static final int PRICE_FIELD = 4;
38         private static final int QOH_FIELD = 5;
39
40         private static final int VALID = 0;
41         private static final int DELETED = 1;
42
43         /**
44          * Field offsets used with String.substring() method
45          * Note: These are not used in the final program!
46          */
47         private static final int DELETED_START_OFFSET = 0;
48         private static final int DELETED_STOP_OFFSET = 1;
49         private static final int TITLE_START_OFFSET = 1;
50         private static final int TITLE_STOP_OFFSET = 51;
51         private static final int AUTHOR_START_OFFSET = 51;
52         private static final int AUTHOR_STOP_OFFSET = 101;
53         private static final int PUB_CODE_START_OFFSET = 101;
54         private static final int PUB_CODE_STOP_OFFSET = 105;
55         private static final int ISBN_START_OFFSET = 105;
56         private static final int ISBN_STOP_OFFSET = 118;
57         private static final int PRICE_START_OFFSET = 118;
58         private static final int PRICE_STOP_OFFSET = 126;
59         private static final int QOH_START_OFFSET = 126;
60         private static final int QOH_STOP_OFFSET = 130;
61
62         /**
63          * Instance Fields
64          */
65
66         private File _filename = null;
67         private RandomAccessFile _raf = null;

```

```

68     private long         _record_count           = 0;
69     private HashMap     _locked_records_map     = null;
70     private Random      _token_maker           = null;
71     private long        _current_record_number = 0;
72
73
74     /**
75     * Constructor
76     *
77     */
78     public DataFileAdapter(String filename) throws InvalidDataFileException {
79         try{
80             _filename = new File(filename);
81             _raf = new RandomAccessFile(_filename, "rw");
82             _raf.seek(0);
83             if((_raf.length() >= HEADER_LENGTH) &&
84                 (_raf.readShort() == FILE_IDENTIFIER)){ // it's a valid data file
85                 System.out.println("Setting up data file for I/O operations");
86                 _record_count = ((_raf.length() - HEADER_LENGTH) / RECORD_LENGTH);
87                 _current_record_number = 0;
88                 _locked_records_map = new HashMap();
89                 _token_maker = new Random();
90             }else if(_raf.length() == 0) { // it's an empty file - make it a data file
91                 _raf.seek(0);
92                 _raf.writeShort(FILE_IDENTIFIER);
93                 _raf.writeBytes(DELETED_STRING);
94                 _raf.writeShort(DELETED_FIELD_LENGTH);
95                 _raf.writeBytes(TITLE_STRING);
96                 _raf.writeShort(TITLE_FIELD_LENGTH);
97                 _raf.writeBytes(AUTHOR_STRING);
98                 _raf.writeShort(AUTHOR_FIELD_LENGTH);
99                 _raf.writeBytes(PUB_CODE_STRING);
100                _raf.writeShort(PUB_CODE_FIELD_LENGTH);
101                _raf.writeBytes(ISBN_STRING);
102                _raf.writeShort(ISBN_FIELD_LENGTH);
103                _raf.writeBytes(PRICE_STRING);
104                _raf.writeShort(PRICE_FIELD_LENGTH);
105                _raf.writeBytes(QOH_STRING);
106                _raf.writeShort(QOH_FIELD_LENGTH);
107
108                _record_count = 0;
109                _locked_records_map = new HashMap();
110                _token_maker = new Random();
111
112            } else {
113                _raf.seek(0);
114                if(_raf.readShort() != FILE_IDENTIFIER){
115                    _raf.close();
116                    System.out.println("Invalid data file. Closing file.");
117                    throw new InvalidDataFileException(null);
118                }
119            }
120
121        }catch(IOException e){
122            System.out.println("Problem opening or creating data file.");
123            System.out.println(e.toString());
124        }
125
126    } // end constructor
127
128
129     /**
130     * Default constructor
131     *
132     */
133     public DataFileAdapter() throws InvalidDataFileException {
134         this("books.dat");
135     }
136
137
138
139     /**
140     * CreateNewDataFile
141     *
142     */
143     public void createNewDataFile(String filename) throws NewDataFileException{
144         try{
145             if(_raf != null){
146                 _raf.close();
147             }
148             _raf = new RandomAccessFile(filename, "rw");

```

```

149
150     _raf.seek(0);
151     _raf.writeShort(FILE_IDENTIFIER);
152     _raf.writeBytes(DELETED_STRING);
153     _raf.writeShort(DELETED_FIELD_LENGTH);
154     _raf.writeBytes(TITLE_STRING);
155     _raf.writeShort(TITLE_FIELD_LENGTH);
156     _raf.writeBytes(AUTHOR_STRING);
157     _raf.writeShort(AUTHOR_FIELD_LENGTH);
158     _raf.writeBytes(PUB_CODE_STRING);
159     _raf.writeShort(PUB_CODE_FIELD_LENGTH);
160     _raf.writeBytes(ISBN_STRING);
161     _raf.writeShort(ISBN_FIELD_LENGTH);
162     _raf.writeBytes(PRICE_STRING);
163     _raf.writeShort(PRICE_FIELD_LENGTH);
164     _raf.writeBytes(QOH_STRING);
165     _raf.writeShort(QOH_FIELD_LENGTH);
166
167     _record_count = 0;
168     _locked_records_map = new HashMap();
169     _token_maker      = new Random();
170
171 }catch(IOException e){
172     System.out.println(e.toString());
173     throw new NewDataFileException(e);
174 }
175
176 } // end createNewDataFile method
177
178
179
180
181 /**
182  * OpenDataFile
183  *
184  */
185 public void openDataFile(String filename) throws InvalidDataFileException {
186     try{
187         if(_raf != null){
188             _raf.close();
189         }
190         _filename = new File(filename);
191         _raf = new RandomAccessFile(_filename, "rw");
192         _raf.seek(0);
193         if((_raf.length() >= HEADER_LENGTH) &&
194             (_raf.readShort() == FILE_IDENTIFIER)){ // it's a valid data file
195             System.out.println("Setting up data file for I/O operations");
196             _record_count = ((_raf.length() - HEADER_LENGTH) / RECORD_LENGTH);
197             _current_record_number = 0;
198             _locked_records_map = new HashMap();
199             _token_maker = new Random();
200         }else if(_raf.length() == 0) { // it's an empty file - make it a data file
201             _raf.seek(0);
202             _raf.writeShort(FILE_IDENTIFIER);
203             _raf.writeBytes(DELETED_STRING);
204             _raf.writeShort(DELETED_FIELD_LENGTH);
205             _raf.writeBytes(TITLE_STRING);
206             _raf.writeShort(TITLE_FIELD_LENGTH);
207             _raf.writeBytes(AUTHOR_STRING);
208             _raf.writeShort(AUTHOR_FIELD_LENGTH);
209             _raf.writeBytes(PUB_CODE_STRING);
210             _raf.writeShort(PUB_CODE_FIELD_LENGTH);
211             _raf.writeBytes(ISBN_STRING);
212             _raf.writeShort(ISBN_FIELD_LENGTH);
213             _raf.writeBytes(PRICE_STRING);
214             _raf.writeShort(PRICE_FIELD_LENGTH);
215             _raf.writeBytes(QOH_STRING);
216             _raf.writeShort(QOH_FIELD_LENGTH);
217
218             _record_count      = 0;
219             _locked_records_map = new HashMap();
220             _token_maker      = new Random();
221
222         } else {
223             _raf.seek(0);
224             if(_raf.readShort() != FILE_IDENTIFIER){
225                 _raf.close();
226                 System.out.println("Invalid data file. Closing file.");
227                 throw new InvalidDataFileException(null);
228             }
229         }

```



```

230
231         }catch(IOException e){
232             System.out.println("Problem opening or creating data file.");
233             System.out.println(e.toString());
234         }
235
236     } // end openDataFile method
237
238
239
240
241 /**
242  * Read the record indicated by the rec_no and return a string array
243  * where each element contains a field value.
244  */
245     public String[] readRecord(long rec_no) throws RecordNotFoundException {
246         String[] temp_string = null;
247         if((rec_no < 0) || (rec_no > _record_count)){
248             System.out.println("Requested record out of range!");
249             throw new RecordNotFoundException("Requested record out of range", null);
250         }else{
251             try{
252                 gotoRecordNumber(rec_no);
253                 if(_raf.readByte() == DELETED){
254                     System.out.println("Record has been deleted!");
255                     throw new RecordNotFoundException("Record " + rec_no + " deleted!", null);
256                 }else{
257                     temp_string = recordBytesToStringArray(rec_no);
258                 }
259             }catch(IOException e){
260                 System.out.println(e.toString());
261                 throw new RecordNotFoundException("Problem in readRecord method", e);
262             }
263         } // end else
264         return temp_string;
265     } // end readRecord()
266
267
268
269
270 /**
271  * Update a record's fields. The record must be locked with the lockRecord()
272  * method and the lock_token must be valid. The value for field n appears in
273  * element record[n]. The call to updateRecord() MUST be preceded by a call
274  * to lockRecord() and followed by a call to unlockRecord()
275  */
276     public void updateRecord(long rec_no, String[] record, long lock_token) throws
277     RecordNotFoundException, SecurityException{
278         if(lock_token != ((Long)_locked_records_map.get(new Long(rec_no))).longValue()){
279             System.out.println("Invalid update record lock token.");
280             throw new SecurityException("Invalid update record lock token.", null);
281         }else{
282             try{
283                 gotoRecordNumber(rec_no); //i.e., goto indicated record
284                 _raf.writeByte((byte)0);
285                 _raf.write(stringToPaddedByteField(record[TITLE_FIELD], TITLE_FIELD_LENGTH));
286                 _raf.write(stringToPaddedByteField(record[AUTHOR_FIELD], AUTHOR_FIELD_LENGTH));
287                 _raf.writeInt(Integer.parseInt(record[PUB_CODE_FIELD]));
288                 _raf.write(stringToPaddedByteField(record[ISBN_FIELD], ISBN_FIELD_LENGTH));
289                 _raf.write(stringToPaddedByteField(record[PRICE_FIELD], PRICE_FIELD_LENGTH));
290                 _raf.writeInt(Integer.parseInt(record[QOH_FIELD]));
291                 _current_record_number = rec_no;
292             }catch(IOException e){
293                 System.out.println("updateRecord(): Problem writing record information.");
294                 System.out.println(e.toString());
295             }
296             catch(NumberFormatException e){
297                 System.out.println("updateRecord(): Problem converting Strings to numbers.");
298                 System.out.println(e.toString());
299             }
300             catch(Exception e){
301                 System.out.println("updateRecord(): Exception occured.");
302                 System.out.println(e.toString());
303             }
304         } // end else
305     } // end updateRecord()
306
307
308
309
310 /**

```

```

311     * Marks a record for deletion by setting the deleted field to 1. The lock_token
312     * must be valid otherwise a SecurityException is thrown.
313     */
314     public void deleteRecord(long rec_no, long lock_token) throws
315     RecordNotFoundException, SecurityException {
316         if(lock_token != ((Long)_locked_records_map.get(new Long(rec_no))).longValue()){
317             System.out.println("Invalid delete record lock token.");
318             throw new SecurityException("Invalid delete record lock token.", null);
319         }else{
320             try{
321                 gotoRecordNumber(rec_no); // goto record indicated
322                 _raf.writeByte((byte)1); // mark for deletion
323             }catch (IOException e){
324                 System.out.println("deleteRecord(): " + e.toString());
325             }
326         }
327     } // end else
328 } // end deleteRecord()
329
330
331
332
333
334 /**
335  * Creates a new datafile record and returns the record number.
336  */
337     public long createRecord(String[] record) throws FailedRecordCreationException {
338         try{
339             gotoRecordNumber(_record_count); //i.e., goto end of file
340             _raf.writeByte((byte)0);
341             _raf.write(stringToPaddedByteField(record[TITLE_FIELD], TITLE_FIELD_LENGTH));
342             _raf.write(stringToPaddedByteField(record[AUTHOR_FIELD], AUTHOR_FIELD_LENGTH));
343             _raf.writeInt(Integer.parseInt(record[PUB_CODE_FIELD]));
344             _raf.write(stringToPaddedByteField(record[ISBN_FIELD], ISBN_FIELD_LENGTH));
345             _raf.write(stringToPaddedByteField(record[PRICE_FIELD], PRICE_FIELD_LENGTH));
346             _raf.writeInt(Integer.parseInt(record[QOH_FIELD]));
347             _current_record_number = ++_record_count;
348         }catch(IOException e){
349             System.out.println(e.toString());
350             throw new FailedRecordCreationException("IOException in createRecord method", e);
351         }
352         catch(NumberFormatException e){
353             System.out.println(e.toString());
354             throw new FailedRecordCreationException("NumberFormatException in createRecord", e);
355         }
356         catch(RecordNotFoundException e){
357             System.out.println(e.toString());
358             throw new FailedRecordCreationException("RecordNotFoundException in createRecord", e);
359         }
360         return _current_record_number;
361     }
362
363
364
365
366 /**
367  * Locks a record for updates and deletes and returns an integer
368  * representing a lock token.
369  */
370     public long lockRecord(long rec_no) throws RecordNotFoundException {
371         long lock_token = 0;
372         if((rec_no < 0) || (rec_no > _record_count)){
373             System.out.println("Record cannot be locked. Not in valid range.");
374             throw new RecordNotFoundException("Record cannot be locked. Not in valid range.", null);
375         }else synchronized(_locked_records_map){
376             while(_locked_records_map.containsKey(new Long(rec_no))){
377                 try{
378                     _locked_records_map.wait();
379                 }catch(InterruptedException ignored){ }
380             }
381             lock_token = _token_maker.nextLong();
382             _locked_records_map.put(new Long(rec_no), new Long(lock_token));
383         }
384
385         return lock_token;
386     } // end lockRecord method
387
388
389
390
391 /**

```

```

392     * Unlocks a previously locked record. The lock_token must be valid or a
393     * SecurityException is thrown.
394     */
395     public void unlockRecord(long rec_no, long lock_token) throws SecurityException {
396         synchronized(_locked_records_map){
397             if(_locked_records_map.containsKey(new Long(rec_no)){
398                 if(lock_token == ((Long)_locked_records_map.get(new Long(rec_no))).longValue()){
399                     _locked_records_map.remove(new Long(rec_no));
400                     _locked_records_map.notifyAll();
401                 }else{
402                     System.out.println("Invalid lock token.");
403                     throw new SecurityException("Invalid lock token", null);
404                 }
405             }else{
406                 System.out.println("Invalid record unlock key.");
407                 throw new SecurityException("Invalid record unlock key.", null);
408             }
409         }
410     } // end unlockRecord method
411
412
413
414
415     /**
416     * Searches the records in the datafile for records that match the String
417     * values of search_criteria. search_criteria[n] contains the search value
418     * applied against field n. Data files can be searched for Title & Author.
419     *
420     */
421     public long[] searchRecords(String[] search_criteria){
422         Vector hit_vector = new Vector();
423         for(int i=0; i<_record_count; i++){
424             try{
425                 if(thereIsAMatch(search_criteria, readRecord(i)){
426                     hit_vector.addElement(new Long(i));
427                 }
428             }catch(RecordNotFoundException ignored){ } // ignore deleted records
429         } // end for
430         long hits[] = new long[hit_vector.size()];
431         for(int i=0; i<hits.length; i++){
432             hits[i] = ((Long)hit_vector.elementAt(i)).longValue();
433         }
434         return hits;
435     }
436
437
438
439
440
441     /**
442     * thereIsAMatch() is a utility method that actually performs
443     * the record search. Implements an implied OR/AND search by detecting
444     * the first character of the Title criteria element.
445     */
446     private boolean thereIsAMatch(String[] search_criteria, String[] record){
447         boolean match_result = false;
448         final int TITLE = 0;
449         final int AUTHOR = 1;
450         for(int i=0; i<search_criteria.length; i++){
451             if((search_criteria[i].length() == 0) || (record[i+1].startsWith(search_criteria[i]))){
452                 match_result = true;
453                 break;
454             } //end if
455         } //end for
456
457         if(((search_criteria[TITLE].length() > 1) && (search_criteria[AUTHOR].length() >= 1)) &&
458             (search_criteria[TITLE].charAt(0) == '&')) {
459             if(record[TITLE+1].startsWith(search_criteria[TITLE].substring(1,
460 search_criteria[TITLE].length()).trim()) && record[AUTHOR+1].startsWith(search_criteria[AUTHOR])){
461                 match_result = true;
462             }else {
463                 match_result = false;
464             }
465         } // end if
466         return match_result;
467     } // end thereIsAMatch()
468
469
470
471

```

```

472     /**
473     * gotoRecordNumber - utility function that handles the messy
474     * details of seeking a particular record.
475     *
476     */
477     private void gotoRecordNumber(long record_number) throws RecordNotFoundException {
478         if((record_number < 0) || (record_number > _record_count)){
479             throw new RecordNotFoundException(null);
480         }else{
481             try{
482                 _raf.seek(RECORDS_START + (record_number * RECORD_LENGTH));
483             }catch(IOException e){
484                 System.out.println(e.toString());
485                 throw new RecordNotFoundException(e);
486             }
487         }
488     } // end else
489
490 } // end gotoRecordNumber method
491
492
493
494 /**
495 * stringToPaddedByteField - pads the field to maintain fixed
496 * field length.
497 *
498 */
499 protected byte[] stringToPaddedByteField(String s, int field_length){
500     byte[] byte_field = new byte[field_length];
501     if(s.length() <= field_length){
502         for(int i = 0; i<s.length(); i++){
503             byte_field[i] = (byte)s.charAt(i);
504         }
505         for(int i = s.length(); i<field_length; i++){
506             byte_field[i] = (byte)' '; //pad the field
507         }
508     }else{
509         for(int i = 0; i<field_length; i++){
510             byte_field[i] = (byte)s.charAt(i);
511         }
512     }
513     return byte_field;
514 } // end stringToPaddedByteField()
515
516
517 /**
518 * recordBytesToStringArray - reads an array of bytes from a data file
519 * and converts them to an array of Strings. The first element of the
520 * returned array is the record number. The length of the byte array
521 * argument is RECORD_LENGTH -1.
522 *
523 */
524 private String[] recordBytesToStringArray(long record_number){
525     String record_string = null;
526     String[] string_array = new String[FIELD_COUNT];
527     byte[] title = new byte[TITLE_FIELD_LENGTH];
528     byte[] author = new byte[AUTHOR_FIELD_LENGTH];
529     byte[] isbn = new byte[ISBN_FIELD_LENGTH];
530     byte[] price = new byte[PRICE_FIELD_LENGTH];
531
532     try{
533         string_array[0] = Long.toString(record_number);
534         _raf.read(title);
535         string_array[TITLE_FIELD + 1] = new String(title).trim();
536         _raf.read(author);
537         string_array[AUTHOR_FIELD + 1] = new String(author).trim();
538         string_array[PUB_CODE_FIELD + 1] = Integer.toString(_raf.readInt());
539         _raf.read(isbn);
540         string_array[ISBN_FIELD + 1] = new String(isbn);
541         _raf.read(price);
542         string_array[PRICE_FIELD + 1] = new String(price).trim();
543         string_array[QOH_FIELD + 1] = Integer.toString(_raf.readInt());
544     }catch(IOException e){
545         System.out.println(e.toString());
546     }
547     return string_array;
548 } // end recordBytesToStringArray()
549
550
551
552 /**

```

```

553     * readHeader - reads the header bytes and converts them to
554     * a string
555     */
556     */
557     public String readHeader(){
558         StringBuffer sb = new StringBuffer();
559         byte[] deleted = new byte[DELETED_STRING.length()];
560         byte[] title = new byte[TITLE_STRING.length()];
561         byte[] author = new byte[AUTHOR_STRING.length()];
562         byte[] pub_code = new byte[PUB_CODE_STRING.length()];
563         byte[] isbn = new byte[ISBN_STRING.length()];
564         byte[] price = new byte[PRICE_STRING.length()];
565         byte[] qoh = new byte[QOH_STRING.length()];
566         try{
567             _raf.seek(0);
568             sb.append(Short.toString(_raf.readShort() + " ");
569             _raf.read(deleted);
570             sb.append(new String(deleted) + " ");
571             sb.append(Short.toString(_raf.readShort() + " ");
572             _raf.read(title);
573             sb.append(new String(title) + " ");
574             sb.append(Short.toString(_raf.readShort() + " ");
575             _raf.read(author);
576             sb.append(new String(author) + " ");
577             sb.append(Short.toString(_raf.readShort() + " ");
578             _raf.read(pub_code);
579             sb.append(new String(pub_code) + " ");
580             sb.append(Short.toString(_raf.readShort() + " ");
581             _raf.read(isbn);
582             sb.append(new String(isbn) + " ");
583             sb.append(Short.toString(_raf.readShort() + " ");
584             _raf.read(price);
585             sb.append(new String(price) + " ");
586             sb.append(Short.toString(_raf.readShort() + " ");
587             _raf.read(qoh);
588             sb.append(new String(qoh) + " ");
589             sb.append(Short.toString(_raf.readShort() + " ");
590         }catch(IOException e){
591             System.out.println(e.toString());
592         }
593         return sb.toString();
594     } // end readHeader()
595
596     /**
597     * getRecordCount() returns the number of records contained in the data file
598     *
599     */
600     public long getRecordCount(){ return _record_count; }
601
602 } // end DataFileAdapter class definition

```

18.24 FailedRecordCreationException.java

```

1     public class FailedRecordCreationException extends Exception {
2
3         /*****
4         * Constructor that takes a String argument
5         *****/
6         public FailedRecordCreationException(String message, Throwable cause){
7             super(message, cause);
8         }
9
10        /*****
11        * Constructor that takes Throwable argument
12        *****/
13        public FailedRecordCreationException(Throwable cause){
14            this("Failed Record Creation Exception", cause);
15        }
16    }
17

```

18.25 InvalidDataFileException.java

```

1     public class InvalidDataFileException extends Exception {
2
3         /*****
4         * Constructor - Takes one String argument
5         *****/
6         public InvalidDataFileException(String message, Throwable cause){
7             super(message, cause);
8         }

```

```

9
10
11      /*****
12      * Constructor that takes Throwable argument
13      *****/
14      public InvalidDataFileException(Throwable cause){
15          this("Invalid Data File Exception", cause);
16      }
17

```

18.26 NewDataFileException.java

```

1      public class NewDataFileException extends Exception {
2
3          /*****
4          * Constructor - One String argument
5          *****/
6          public NewDataFileException(String message, Throwable cause){
7              super(message, cause);
8          }
9
10
11         /*****
12         * Constructor that takes Throwable argument
13         *****/
14         public NewDataFileException(Throwable cause){
15             this("New Data File Exception", cause);
16         }
17

```

18.27 RecordNotFoundException.java

```

1      public class RecordNotFoundException extends Exception {
2
3          /*****
4          * Constructor - Takes one String argument
5          *****/
6          public RecordNotFoundException(String message, Throwable cause){
7              super(message, cause);
8          }
9
10         /*****
11         * Constructor that takes Throwable argument
12         *****/
13         public RecordNotFoundException(Throwable cause){
14             this("Record Not Found Exception", cause);
15         }
16

```

18.28 SecurityException.java

```

1      public class SecurityException extends Exception {
2
3          /*****
4          * Constructor - Takes one String argument
5          *****/
6          public SecurityException(String message, Throwable cause){
7              super(message, cause);
8          }
9
10         /*****
11         * Constructor that takes a Throwable argument
12         *****/
13         public SecurityException(Throwable cause){
14             this("Security Exception", cause);
15         }
16

```

18.29 AdapterTesterApp.java

```

1      public class AdapterTesterApp {
2          public static void main(String[] args){
3              try{
4                  DataFileAdapter adapter = new DataFileAdapter();
5                  String[] rec_1 = {"C++ For Artists", "Rick Miller", "23", "1-932504-02-8", "$59.95", "80"};
6                  String[] rec_2 = {"Java For Artists", "Rick Miller", "23", "1-932504-04-X", "$69.95", "100"};
7
8                  String[] search_string = {"Java", " "};
9
10                 String[] temp_string = null;
11
12                 //adapter.createNewDataFile("books.dat");

```

```

13         adapter.createRecord(rec_1);
14         adapter.createRecord(rec_1);
15         adapter.createRecord(rec_1);
16         adapter.createRecord(rec_1);
17         adapter.createRecord(rec_1);
18         adapter.createRecord(rec_1);
19         adapter.createRecord(rec_1);
20         adapter.createRecord(rec_1);
21
22         long lock_token = adapter.lockRecord(2);
23
24         adapter.updateRecord(2, rec_2, lock_token);
25         adapter.unlockRecord(2, lock_token);
26
27         lock_token = adapter.lockRecord(1);
28         adapter.deleteRecord(1, lock_token);
29         adapter.unlockRecord(1, lock_token);
30
31
32
33         long[] search_hits = adapter.searchRecords(search_string);
34
35
36         System.out.println(adapter.readHeader());
37
38         for(int i=0; i<search_hits.length; i++){
39             try{
40                 temp_string = adapter.readRecord(search_hits[i]);
41                 for(int j = 0; j<temp_string.length; j++){
42                     System.out.print(temp_string[j]+" ");
43                 }
44                 System.out.println();
45             }catch(RecordNotFoundException e){ System.out.println(e.toString()); }
46         }
47     }catch(Exception e){ e.printStackTrace(); }
48 }
49
50 }
51 }

```

Referring to example 18.29 — the AdapterTesterApp class tests the functionality of the DataFileAdapter class and its supporting classes. A DataFileAdapter object is created on line 4. On lines 5 and 6 two String arrays are created that contain the field values for records that will be inserted into the legacy data file for testing purposes. A String array named search_string is created on line 8. This array will be used to test the searchRecords() method.

The statement on line 12 is commented out, but was initially used to test the createNewDataFile() method. Lines 13 through 20 insert a series of record values using the data contained in the rec_1 array.

On line 22 the lockRecord() method is tested and the lock_token obtained is used on lines 24 and 25 to test the updateRecord() and unlockRecord() methods.

The searchRecords() method is tested on line 33 and the resulting array of longs is used in the for statement beginning on line 38 to print the record data for each record returned in the search. Figure 18-30 shows the results of running this program. It should be noted that each time the program is executed in its current version additional records with the data contained in the rec_1 array will be appended to the end of the file.

```

Terminal — tcsh
[Rick-Millers-Computer: ~/desktop/randomaccessfiletest] swodog$ java AdapterTesterApp
Setting up data file for I/O operations
Record has been deleted!
378 deleted 1 title 50 author 50 pub_code 4 ISBN 13 price 8 qoh 4
2 Java For Artists Rick Miller 23 1-932504-04-X $69.95 100
[Rick-Millers-Computer: ~/desktop/randomaccessfiletest] swodog$

```

Figure 18-30: Results of Running Example 18.29.

Quick Review

The RandomAccessFile class is used to write and read bytes, characters, and primitive type values to and from any position within a file. The RandomAccessFile stands alone in that it is not related to the InputStream, OutputStream, Reader, or Writer classes.

Quick File I/O COMBINATION REFERENCE

Don't sweat the load too much if you still feel somewhat dazed and confused after reading this chapter. The only way you get good at this stuff is through experience. Trust me when I say that you are not alone in your confusion. To help ease your pain and speed your understanding I have included table 18-3 which lists various combinations of the java.io classes based on intended usage.

If you want to...	Use the following class or class combination...
<i>Byte-Oriented Stream Output</i>	
Write long streams of bytes	FileOutputStream
Write long streams of bytes with improved output efficiency	FileOutputStream BufferedOutputStream
Write streams of bytes and primitive type values	FileOutputStream DataOutputStream
Write streams of bytes and primitive types with improved efficiency	FileOutputStream BufferedOutputStream DataOutputStream
Write serialized objects	FileOutputStream ObjectOutputStream
Write serialized objects with improved efficiency	FileOutputStream BufferedOutputStream ObjectOutputStream
Write textual representations of primitive types and objects	FileOutputStream PrintStream
Write textual representation of primitive types and objects with improved efficiency	FileOutputStream BufferedOutputStream PrintStream
<i>Byte-Oriented Stream Input</i>	
Read long streams of bytes	FileInputStream
Read long streams of bytes with improved efficiency	FileInputStream BufferedInputStream
Read serialized objects from disk	FileInputStream ObjectInputStream
Read serialized objects from disk with improved efficiency	FileInputStream BufferedInputStream ObjectInputStream
<i>Character-Oriented Stream Output</i>	
Write text to disk in default local encoding	FileWriter

Table 18-3: Handy java.io Class Combination Reference

If you want to...	Use the following class or class combination...
Write text to disk in default local encoding with improved efficiency	FileWriter BufferedWriter
Write characters to disk with more control over character encoding scheme	FileOutputStream OutputStreamWriter
Write characters to disk with more control over character encoding and with improved efficiency	FileOutputStream OutputStreamWriter BufferedWriter
Write textual representations of primitive types and objects with control over character encoding and with improved efficiency	FileOutputStream OutputStreamWriter BufferedOutputStream PrintWriter
<i>Character-Oriented Stream Input</i>	
Read text from file in default local encoding	FileReader
Read text from file in default local encoding with improved efficiency	FileReader BufferedReader
Read text from file with more control over character encoding	FileInputStream InputStreamReader
Read text from file with more control over character encoding and with improved efficiency	FileInputStream InputStreamReader BufferedReader
<i>Random Access File Input and Output</i>	
Read and write bytes, characters, and primitive type values anywhere within a file	RandomAccessFile

Table 18-3: Handy java.io Class Combination Reference

SUMMARY

The classes in the java.io package can be used to perform I/O operations on various types of data sinks and sources. A sink is a destination to which data is written, and a source is an origination point from which data is read. Strings, arrays, processes, and files are examples of the types of sinks and sources supported by the java.io classes. This chapter focused on using the java.io classes to perform I/O operations on files.

The java.io package can be intimidating to novice Java programmers. But there are several ways to organize the classes in the java.io package to help tame the conceptual complexity. A class hierarchy organization gives valuable clues regarding the intended use of the java.io classes.

It is also helpful to categorize java.io classes into file-terminal, intermediate, and user-fronting categories. File-terminal classes are used to create or open a file for input, output, or input/output operations. There are five file-terminal classes: FileInputStream, FileOutputStream, FileReader, FileWriter, and RandomAccessFile. Intermediate classes are used to enhance the performance of file terminal classes. Generally speaking, any class that contains the word Buffer is an intermediate class. User-fronting classes are those classes that have a wide variety of user interface methods that make byte, character, primitive type, and/or object I/O operations convenient for the programmer.

The File class represents the name of a file or directory and is used to manipulate file or directory metadata such as a file's name, length, path, or the list of files in a directory, etc. The creation of a File object does not result in an actual file being created on disk, however, you can create an empty file by calling the createNewFile() method.

The `InputStream` and `OutputStream` classes operate on byte streams. With `InputStreams` and `OutputStreams` you can read and write bytes, primitive types, 8-bit characters, and serialized objects. When performing character I/O with byte streams the local default character encoding is utilized.

The `Reader` and `Writer` classes operate on 16-bit Unicode characters. The character encoding can be specified. This is an important internationalization feature.

The `RandomAccessFile` class is used to perform both file input and output on bytes, chars, and primitive types.

Skill-Building Exercises

1. **Web Research:** In this chapter I emphasized how the `Reader` and `Writer` classes are more flexible with regards to character I/O because they allow you to specify the character encoding scheme used to translate the bytes stored on file to characters used in your program. Utilizing the Web, research the topic of Java character encoding. Specifically, determine the default local encoding for your computer and what encoding schemes your computer's operating system supports.
2. **API Exploration:** Research the `java.nio` package and compare its functionality to that provided by the `java.io` package.
3. **Code Drill:** Compile and run each of the example programs listed in this chapter. Make changes to the programs to change the data written to the files. Expand the programs to utilize more of the interface methods offered by each of the `java.io` classes demonstrated.
4. **API Exploration:** Study the complete `java.io` package. Note the similarities between the classes that were discussed in this chapter and those that were omitted. Concentrate especially on those classes that read or write from other data sources and sinks to include `Strings`, `arrays`, and `processes`.
5. **Programming:** Write a short program to explore the effects of writing and reading various kinds of `Serializable` objects to and from a file.
6. **API Exploration:** Study the classes belonging to the `java.text` package. These classes can come in handy when manipulating text in your programs.
7. **Computer I/O Sub-System Research:** Obtain a book on operating systems and computer hardware systems and research the underlying principles and technologies that support modern computer file I/O operations. Pay particular attention to how computer systems implement `Direct Memory Access (DMA)` and `I/O buffering`.
8. **Programming:** Write a short program and experiment with the different methods provided by the `PrintStream` and `PrintWriter` classes.
9. **Programming:** Write a short program that converts text stored in a file to different Unicode character encodings. Utilize the knowledge gained from skill-building Exercise #1 to help you in this endeavor.
10. **Programming:** Write a short program that exercises the capabilities of the `RandomAccessFile` class.

Suggested Projects

1. **Computer Simulator Revisited:** Redesign and implement the computer simulator project originally presented in chapter 8, suggested project #3. Add the capability to store programs to a file and load programs from a file.

2. **Robot Rat Revisited:** Redesign and implement the Robot Rat program presented in chapter 3. Add the capability to capture the robot rat's movement history to a file and replay the movements stored in a file.
3. **Aircraft Engine Simulation Revisited:** Redesign and implement the aircraft engine simulation presented in chapter 11. Add the capability to store various engine configurations in a file and create engines based upon those engine configuration files.
4. **Tanker Fuel-Oil Pumping System Revisited:** Redesign and implement the tanker pumping system originally presented in chapter 11, suggested project #6. Add the capability to script the filling or draining of tanks. Scripts should be saved to a file for later use.
5. **FileListener:** Write a program that listens for I/O events on a file. Your program might consist of a user-defined interface named FileIOListener. A class that is capable of responding to file I/O events would implement the FileIOListener interface. Study the aircraft engine simulation presented in chapter 11 for ideas on how to implement custom event processing.

Hint: You might proceed by extending one of the java.io classes presented in this chapter and giving it the capability to maintain a list of FileIOListeners.
6. **Encryption:** You and a friend would like to send secret messages to each other. Write a program that reads a text file and encrypts its contents and saves it to a new encrypted file. Explore the capabilities of the javax.crypto package to help you in this effort.
7. **Manipulating Zip Files:** Write a program that lets you select, with a GUI, a set of files that can be gathered together in a zip file. Use the classes in the java.util.zip package to help you with this effort.
8. **Employee Example Revisited:** Revisit the Employee example used in chapter 11 and add the capability to save and retrieve employee information to and from a file.
9. **Home Inventory Database:** Write a program that lets you document the contents of your home and preserve the items in a file for safe keeping. Give your program a GUI front end and the capability to sort the items by type after entry. Also provide the capability to print the list to a printer.
10. **CD Collection Database:** Write a program that lets you document your CD collection. Your program should save the CD list to a file and load the file for future editing.

SELF-TEST QUESTIONS

1. What is the purpose of the File class? Can the File class be used to create an actual file? If so, how?
2. What are the primary differences between the byte-stream classes and the character-stream classes?
3. What is meant by the term data *sink*? What is meant by the term data *source*?
4. What's the purpose of any java.io class that contains the word Buffer in its name? What is the default buffer size for any such class?
5. In your own words describe what is meant by the terms *file terminal class*, *intermediate class*, and *user fronting class*.
6. (True/False) The RandomAccessFile class can be used to write serializable objects to a file.

7. What combination of classes can be used to write and read serializable objects to and from a file?
8. What classes can be used to write textual representations of primitive types and objects to a file?
9. What interface must an object implement before it can be serialized to a file?
10. What classes in the java.io package implement the DataInput or DataOutput interface?

REFERENCES

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*, 4th Edition. O'Reilly and Associates, Inc. Sebastopol, CA. ISBN: 0-596-00283-1

James Gosling, et. al. *The Java™ Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, MA. ISBN: 0-201-31008-2

Jon Meyer, et. al. *Java™ Virtual Machine*. O'Reilly and Associates, Inc. Sebastopol, CA. ISBN: 1-56592-194-1

The Java 2 Standard Edition Version 5 Platform API Online Documentation [<http://java.sun.com/j2se/1.5.0/docs/api/index.html>]

The Java 2 Standard Edition Version 1.4.2 Platform API Online Documentation [<http://java.sun.com/j2se/1.4.2/docs/api/index.html>]

NOTES

PART V: NETWORK PROGRAMMING

CHAPTER 19



Rick

INTRODUCTION TO NETWORKING AND DISTRIBUTED APPLICATIONS

LEARNING OBJECTIVES

- *DEMONSTRATE YOUR UNDERSTANDING OF BASIC NETWORKING CONCEPTS*
- *STATE THE DEFINITION OF THE TERM SERVER HARDWARE*
- *STATE THE DEFINITION OF THE TERM SERVER APPLICATION*
- *DESCRIBE THE DIFFERENCE BETWEEN SERVER HARDWARE AND A SERVER APPLICATION*
- *STATE THE DEFINITION OF THE TERM CLIENT APPLICATION*
- *LIST AND DESCRIBE THE DIFFERENT WAYS IN WHICH SERVER APPLICATIONS AND CLIENT APPLICATIONS CAN BE PHYSICALLY AND LOGICALLY DISTRIBUTED*
- *DESCRIBE THE PROPERTIES OF A MULTI-TIERED APPLICATION*
- *STATE THE DEFINITION OF THE TERMS PROTOCOL, PORT, PACKET, DATAGRAM, TCP/IP AND UDP*
- *DEMONSTRATE YOUR UNDERSTANDING OF REMOTE METHOD INVOCATION (RMI)*
- *DEMONSTRATE YOUR UNDERSTANDING OF HOW INPUTSTREAMS AND OUTPUTSTREAMS ARE USED TO TRANSFER DATA BETWEEN NETWORK APPLICATIONS*
- *DESCRIBE THE PURPOSE AND USE OF SOCKETS*

INTRODUCTION

Network applications are everywhere in today's modern computing environment. If you use email, a Web browser, or a chat program like AOL Instant Messenger, you are using software applications powered by network technology.

This chapter serves two primary purposes. First, it gives you a broad understanding of key networking concepts and terminology. Here you will learn the difference between server software and server hardware, the meanings of the terms network, socket, port, packet, datagram, TCP/IP and UDP, how `InputStreams` and `OutputStreams` are used to transfer data between network applications, and how applications can be physically and logically distributed in a network environment. You will learn how client applications logically interact with server applications. And finally, you will learn the concepts associated with calling methods on an object across a network. This is referred to as Remote Method Invocation (RMI).

The second purpose of this chapter is to introduce you to the concepts of multi-tiered, distributed applications. Modern network applications are often logically tiered with one or more of the logical tiers physically deployed on different computers. It will be important for you to understand the terminology associated with these concepts as you learn to write network-enabled Java applications.

Upon completion of this chapter you will have a solid foundation upon which to successfully approach chapters 20 - Client-Server Applications, and 21 - Applets and JDBC. This chapter is not, however, a compendium on the topic of network programming or distributed applications. The subject is much too rich to adequately cover in one chapter and is quite beyond the scope of this book. If you are interested in pursuing something you learn here in more detail then I recommend you select one of the excellent sources listed in the references section and follow your interests. An excellent place to learn more about Internet programming is the Internet FAQ Archives: [<http://www.faqs.org/faqs/>]

WHAT IS A COMPUTER NETWORK?

A computer network is an interconnected collection of computing devices. A computing device, for the purposes of this rather broad definition, can be any piece of equipment that exists to participate in or support a network in some fashion. Examples of computing devices include general purpose computers, special purpose computers, routers, switches, hubs, printers, personal digital assistants (*PDA*s), etc.

PURPOSE OF A NETWORK

Computer networks are built with a specific purpose in mind. The primary purpose of a computer network is resource sharing. A resource can be physical (*i.e. a printer or a computer*) or metaphysical (*i.e. knowledge or data*). Figure 19-1 shows a diagram for a simple computer network. This type of simple network is referred to as a Local Area Network (LAN).

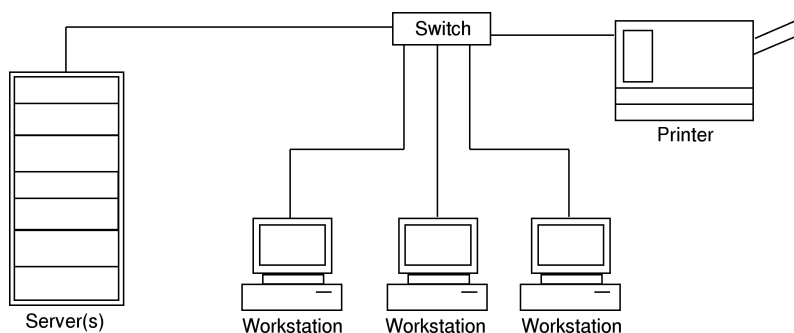


Figure 19-1: A Simple Computer Network

Referring to figure 19-1 — the computing devices participating in this simple network include the workstations, the servers, the printer, and the switch. The switch facilitates network interconnection. In this configuration the workstations and servers can share the computational resources offered by each computer on the network as well as the printing services offered by the printer. Data can also be offered up for sharing on each computer as well.

THE ROLE OF NETWORK PROTOCOLS

A protocol is a specification of rules that govern the conduct of a particular activity. Entities that implement the protocol(s) for a given activity can participate in that activity. For example, Robert's Rules of Order specify a set of protocols for efficiently and effectively conducting meetings. A similar analogy applies to computer networking.

HOMOGENEOUS Vs. HETEROGENEOUS NETWORKS

Computers participating in a computer network communicate with each other via a set of networking protocols. There are generally two types of network environments: 1) *homogeneous* - where all the computers are built by the same company and can talk to each other via that company's proprietary networking protocol, or 2) *heterogeneous* - where the computers are built by different companies, have different operating systems, and therefore different proprietary networking protocols. An example of a homogenous network would be one comprised entirely of Apple Macintosh computers and Apple peripherals. The Macintosh computers could communicate perfectly fine with each other via AppleTalk which is an Apple networking protocol. In a perfect world, we would all use Apple Macintosh computers but the world is, alas, imperfect, and almost every network in existence is heterogeneous in nature. Apples running OS X must communicate with computers running Sun Solaris, Microsoft Windows, Linux, and a host of other hardware and operating system combinations.

THE UNIFYING NETWORK PROTOCOLS: TCP/IP

In today's heterogeneous computer network environment the protocols that power the Internet — Transmission Control Protocol (TCP) and Internet Protocol (IP) — collectively referred to as TCP/IP, have emerged as the standard network protocols through which different types of computers can talk to each other. Figure 19-2 shows the local area network connected to the Internet via a router. So long as the computers on the LAN utilize an operating system that implements TCP/IP then they can access the computational and data resources made available both internally and via the Internet. (*If the LAN does not utilize TCP/IP then a bridge or gateway device would be required to perform the necessary internetwork protocol translation.*)

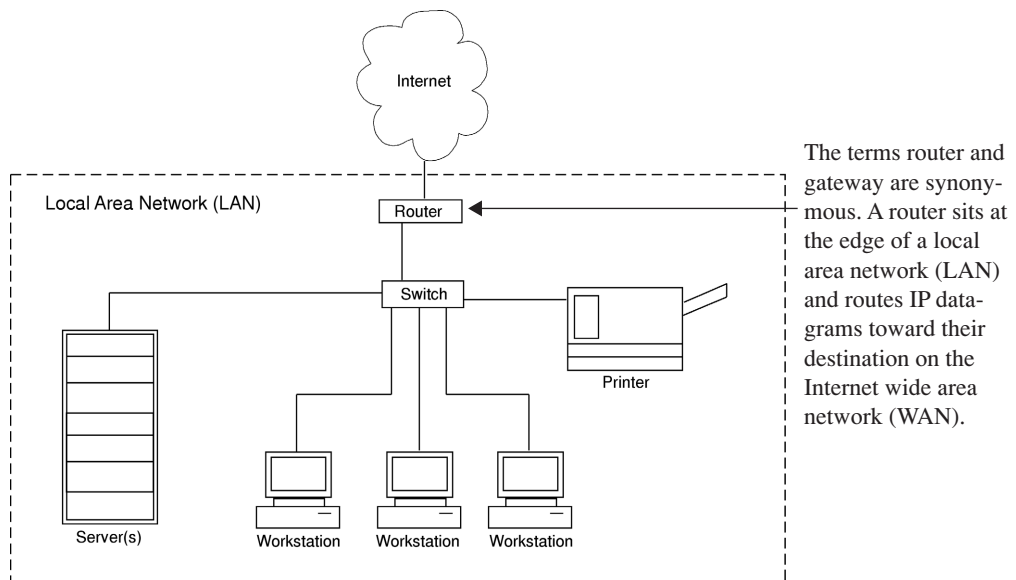


Figure 19-2: Local Area Network Connected to the Internet

WHAT'S SO SPECIAL ABOUT THE INTERNET?

What makes the Internet so special? The answer — TCP/IP. The Internet is a vast network of computer networks. All the networks on the Internet communicate with each other via TCP/IP. The TCP/IP protocols were developed with Department of Defense (DoD) funding. What the DoD wanted was a computer and communications network that was resilient to attack. If a piece of the Internet was destroyed by a nuclear blast then data would be automatically routed through the surviving network connections. When one computer communicates with another computer via the Internet the data it sends is separated into packets and transmitted a packet at a time to the designated computer. TCP/IP provides for packet routing and guaranteed packet delivery. Because of the functionality provided by the TCP/IP protocols the Internet is considered to be a robust and reliable way to transmit and receive data. Figure 19-3 shows how the simple network of agency A can share resources with other agencies via the Internet.

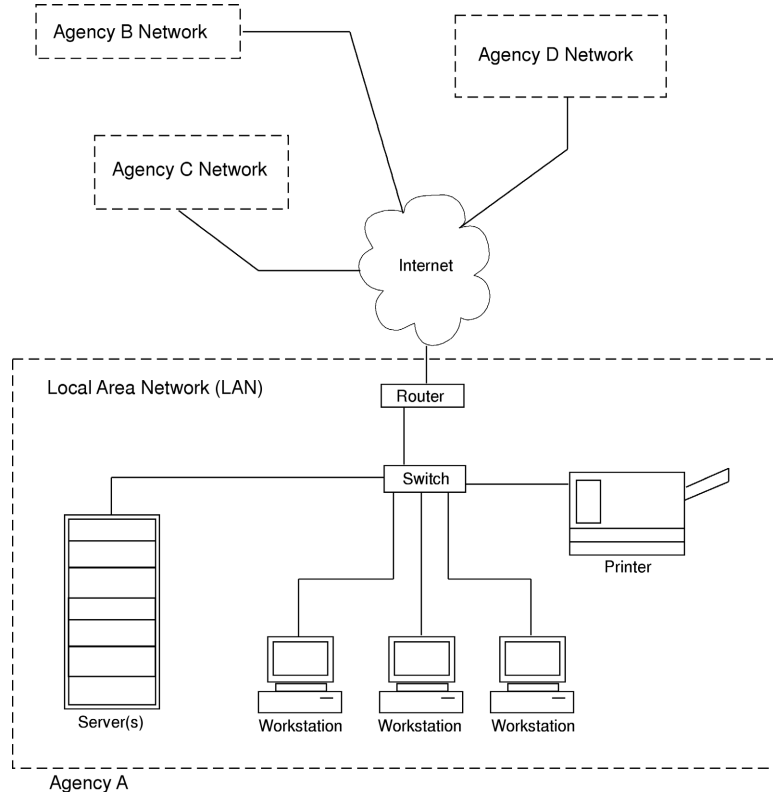


Figure 19-3: The Internet — A Network of Networks Communicating via Internet Protocols

Quick Review

A computer network is an interconnected collection of computing devices. Examples of computing devices include general purpose computers, special purpose computers, routers, switches, hubs, printers, personal digital assistants (*PDA*s), etc. The primary purpose of a computer network is resource sharing. A resource can be physical (*i.e.* a printer) or metaphysical (*i.e.* data).

A protocol is a specification of rules that govern the conduct of a particular activity. Entities that implement the protocol(s) for a given activity can participate in that activity. Computers participating in a computer network communicate with each other via a set of networking protocols. There are generally two types of network environments: 1) *homogeneous* - where all the computers are built by the same company and can talk to each other via that company's proprietary networking protocol, or 2) *heterogeneous* - where the computers are built by different companies, have different operating systems, and therefore different proprietary networking protocols. In today's heterogeneous computer network environment the protocols that power the Internet — Transmission Control Protocol (TCP) and Internet Protocol (IP) — collectively referred to as TCP/IP, have emerged as the standard network protocols through which different types of computers can talk to each other.

What makes the Internet so special? The answer — TCP/IP. When one computer communicates with another computer via the Internet the data it sends is separated into packets and transmitted a packet at a time to the designated computer. TCP/IP provides for packet routing and guaranteed packet delivery. Because of the functionality provided by the TCP/IP protocols the Internet is considered to be a robust and reliable way to transmit and receive data.

SERVERS & CLIENTS

The terms *server* and *client* each have both a hardware and software aspect. This section briefly discusses these terms in both aspects in greater detail to provide you with a foundation for the material presented in the next section.

SERVER HARDWARE AND SOFTWARE

The term *server* is often used to refer both to a piece of computing hardware on which a *server application* runs and to the server application itself. I will use the term server to refer to hardware. I will use the term server application to refer to a software component whose job is to provide some level of service to another entity.

As figure 19-4 illustrates, it is the job of a server to host server applications. However, as desktop computing power increases, the lines between client and server hardware become increasingly blurry. A good definition for a server then is any computer used to host one or more server applications as its primary job. A server is usually (*should be*) treated as a critical piece of capital equipment within an organization. Server operating requirements are used to specify air conditioning, electrical, and flooring requirements for data centers. Servers are supported by data backup and recovery procedures and, if they are truly agency critical, will have some form of fault tolerance and redundancy designed in as well.

A server running a server application is also referred to as a *host*. The term host extends to any computer running any application.

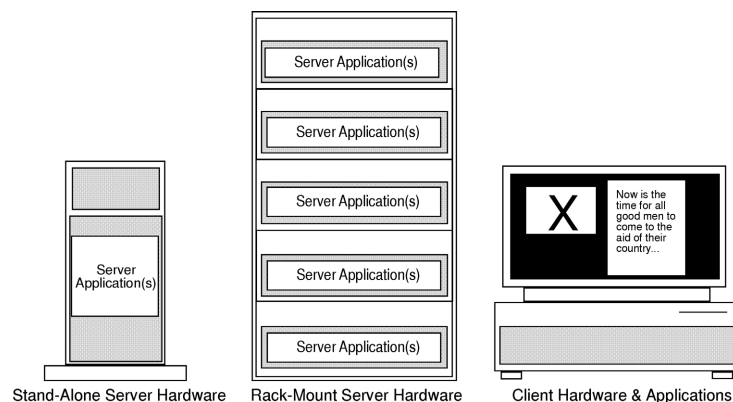


Figure 19-4: Client and Server Hardware and Applications

CLIENT HARDWARE AND SOFTWARE

The term *client* is also used to describe both hardware and software. Client hardware is any computing device that hosts an application that requires or uses the services of a server application. Client software is any application that requires or uses the services provided by a server application. For example, when you run Microsoft Internet Explorer on your home computer you are running a client application. You use Internet Explorer to access web sites via the Internet. These web sites are served up by a web server (i.e., an *HTTP server*) which is a server application hosted on a server somewhere out there in Internet land.

Quick Review

The terms *server* and *client* each have both a hardware and software aspect. The term *server* is often used to refer both to a piece of computing hardware on which a *server application* runs and to the server application itself. A good definition for a server then is any computer used to run one or more server applications as its primary job. A server running a server application is also referred to as a *host*. The term *host* extends to any computer running any application. The term *client* is also used to describe both hardware and software. Client hardware is any computing device that hosts an application that requires or uses the services of a server application. Client software is any application that requires or uses the services provided by a server application.

Application Distribution

The term *application distribution* refers to where (*i.e., on what physical computer*) one or more pieces of a network application reside. This section discusses the concepts of physically distributing client and server applications. Server applications themselves can be further divided into multiple application layers with each distinct application layer being physically deployed to one or more computers. The concepts associated with multi-layered applications are presented and discussed in the next section.

PHYSICAL DISTRIBUTION ON ONE COMPUTER

Client and server applications can both be deployed on the same computer. This is most often done for the purposes of testing during development. When you write applets and client-server applications in chapters 20 and 21 you will test them on your development machine. If you are fortunate enough to have a home network that includes multiple computers you can test your client-server applications in a more real-world setting. Figure 19-5 illustrates the concept of running client and server applications on the same physical hardware.

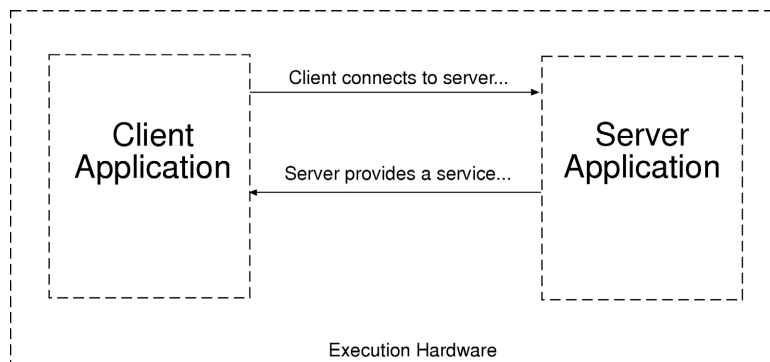


Figure 19-5: Client and Server Applications Physically Deployed to Same Computer

RUNNING MULTIPLE JAVA VIRTUAL MACHINES ON THE SAME COMPUTER

When writing and testing client-server applications using Java you will need to run the server application and client application in separate Java Virtual Machines (*JVMs*). Figure 19-6 illustrates the concept of running multiple JVM instances on the same computer.

STARTING MULTIPLE JVMs IN WINDOWS

When programming Java applications in the Microsoft Windows environment you will use the MS-DOS command prompt emulator application extensively to compile and test your programs. You can launch separate instances of the JVM in their own command prompt windows by using the *start* command at the command line:

```
start java <application class name>
```

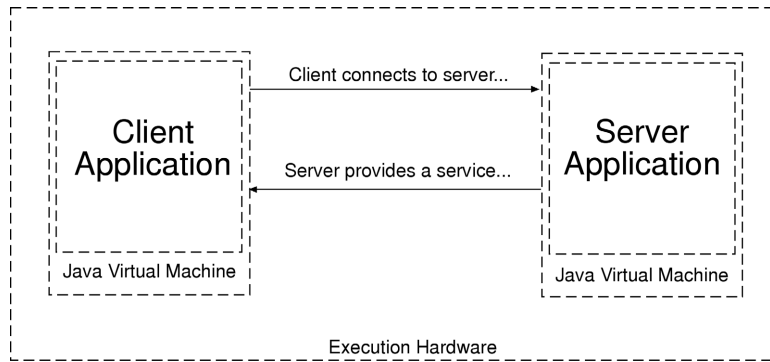


Figure 19-6: Client and Server Applications Require Separate Java Virtual Machines

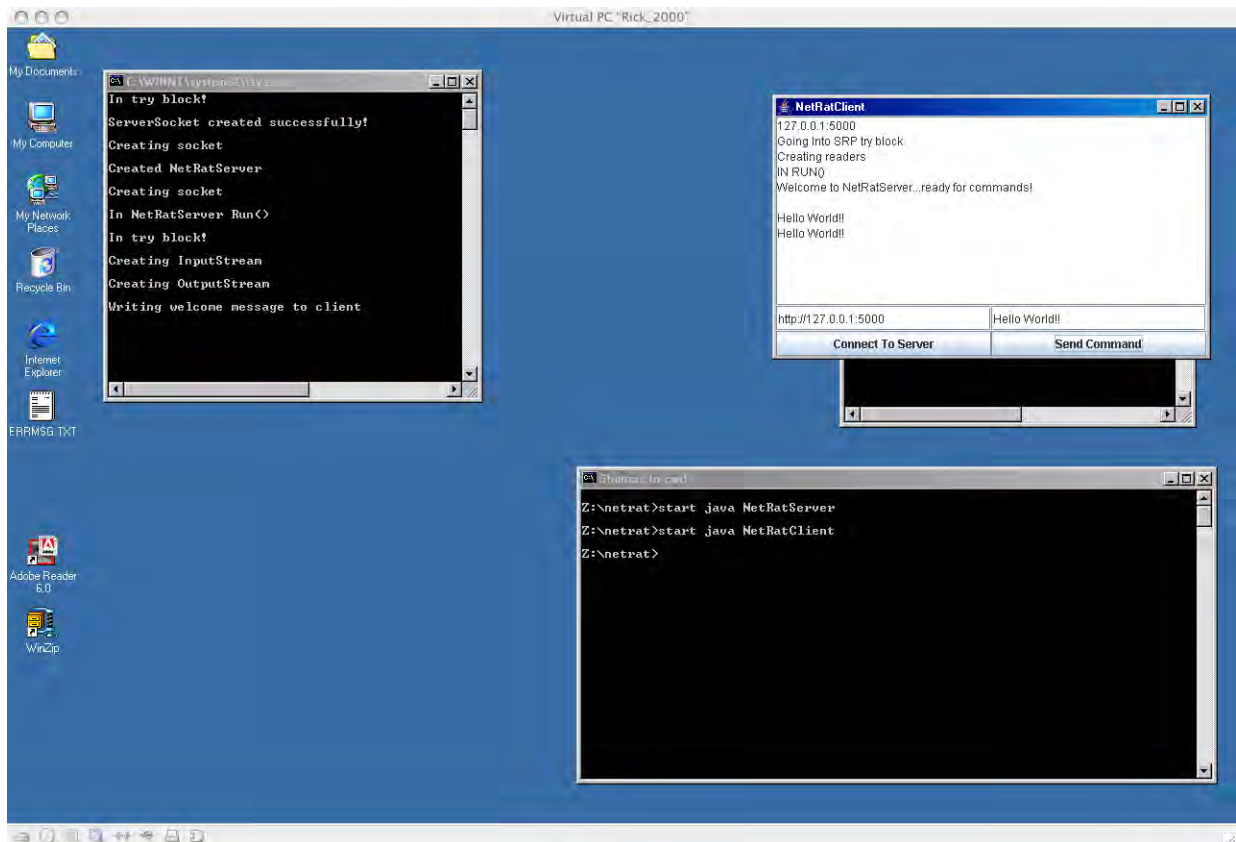


Figure 19-7: Starting Multiple Terminal Windows Using start Command

Figure 19-7 shows multiple JVMs being started in separate command prompt windows using the start command. In this example the server application is named NetRatServer and the client application is named NetRatClient. The start command is shown being issued in the lower-right window titled “Shortcut to cmd”. The NetRatServer application is started first and is shown running in the upper-left command prompt window. The NetRatClient application is running in the upper-right window which is partially obscured by its GUI interface.

STARTING MULTIPLE JVMs IN UNIX AND LINUX

You can start multiple JVMs in Unix or Linux operating systems as well. You can do this in two ways. Either you can open multiple terminal windows and launch each application in a dedicated window or launch each application as a separate process using ‘&’. Figure 19-8 shows the NetRatServer and NetRatClient applications being launched in a Macintosh OS X terminal window as separate processes using the ‘&’.

The screenshot shows a terminal window titled "Terminal — tcsh" with the following output:

```
[Rick-Millers-Computer:itp_120/projects/netrat] swodog% java NetRatServer &
[1] 464
[Rick-Millers-Computer:itp_120/projects/netrat] swodog% In try block!

ServerSocket created successfully!

Creating socket

ps
  PID TT STAT      TIME COMMAND
  455 std S      0:00.12 -tcsh
  464 std R      0:00.45 java NetRatServer
[Rick-Millers-Computer:itp_120/projects/netrat] swodog% java NetRatClient &
[2] 466
[Rick-Millers-Computer:itp_120/projects/netrat] swodog% ps
  PID TT STAT      TIME COMMAND
  455 std S      0:00.13 -tcsh
  464 std S      0:00.53 java NetRatServer
  466 std S      0:02.31 java NetRatClient
[Rick-Millers-Computer:itp_120/projects/netrat] swodog% Created NetRatServer

Creating socket

In NetRatServer Run()

In try block!

Creating InputStream

Creating OutputStream

Writing welcome message to client

[]
```

Annotations on the right side of the terminal window point to specific lines:

- "server application started" points to the first `java NetRatServer &` command.
- "ps command shows processes" points to the first `ps` command.
- "client application started" points to the `java NetRatClient &` command.
- "ps command shows processes" points to the second `ps` command.

Below the terminal window is a window titled "NetRatClient" with the following content:

```
127.0.0.1:5000
Going Into SRP try block
Creating readers
IN RUN()
Welcome to NetRatServer...ready for commands!

http://127.0.0.1:5000      222

[Connect To Server] [Send Command]
```

Figure 19-8: Multiple JVMs Launched As Separate Processes In Mac OSX

Referring to figure 19-8 — One terminal window titled “Terminal — tcsh” is open. The first command launches the NetRatServer application as a separate process. Notice the ‘&’ applied at the end of the command line before hitting the return key. The following line shows a process number in brackets followed by a process number: [1] 464. The next several lines are the result of the server application start-up process.

Using The ps Command To List Processes In UNIX

The `ps` command is used to list currently running processes. After the NetRatServer application is launched the `ps` command is issued to list the processes. The NetRatClient application is then launched as another separate process and again the `ps` command is used to list the processes. As you can see from figure 19-8 both the server and client applications are now running as separate processes.

Killing PROCESSES

When you start multiple JVM instances as separate processes closing the terminal window is not enough to kill the process. You will have to issue the `kill` command explicitly. Figure 19-9 shows the `kill` command being used to kill both the NetRatServer and NetRatClient processes.

The screenshot shows a terminal window titled "Terminal — tcsh" with the following output:

```
Last login: Sat Nov 27 10:22:48 on ttys1
Welcome to Darwin!
[Rick-Millers-Computer:~] swodog% ps
  PID TT STAT      TIME COMMAND
  464 std- S      0:01.31 java NetRatServer
  466 std- S      0:13.48 java NetRatClient
  473 std S      0:00.05 -tcsh
[Rick-Millers-Computer:~] swodog% kill 464
[Rick-Millers-Computer:~] swodog% kill 466
[Rick-Millers-Computer:~] swodog% ps
  PID TT STAT      TIME COMMAND
  473 std S      0:00.05 -tcsh
[Rick-Millers-Computer:~] swodog% []
```

Figure 19-9: Killing Unix Processes with the kill Command

Running Multiple Clients On The Same Computer

You can run multiple client applications on the same computer. Each client will require its own JVM as is shown in figure 19-10. Your server application must be capable of handling multiple concurrent client requests for service. A server application with this capability is generally referred to as being *multi-threaded*. Each incoming client connection is passed off to a unique thread for processing. The execution of multiple client applications, in addition to the server application, on the same hardware, is common practice during a software project's development and testing phases.

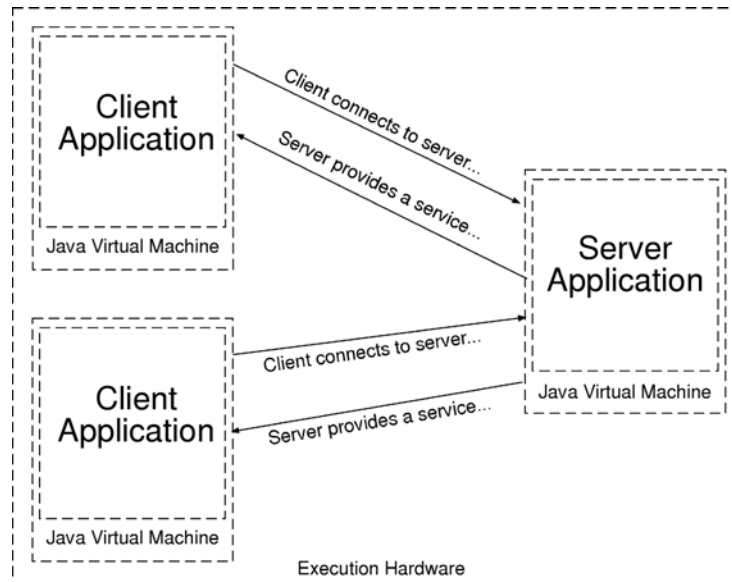


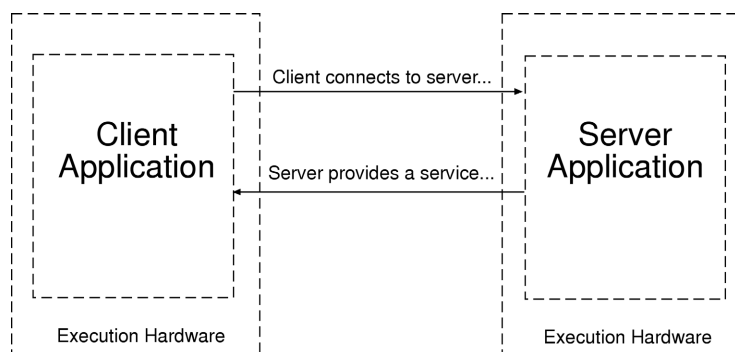
Figure 19-10: Running Multiple Client JVMs On Same Hardware

Addressing The Local Machine

When testing client-server applications on your local machine you can use the localhost IP address of 127.0.0.1 as the server application's host address.

Physical Distribution Across Multiple Computers

Although client and server applications can be co-located on the same hardware it is more often the case that they are physically deployed on different machines geographically separated by great distance. Figure 19-11 illustrates this concept.



These computers may be in the same room on the same local-area-network (LAN) or each may be located half-way around the planet from the other connected via the Internet.

Figure 19-11: Client and Server Applications Deployed On Different Computers

Multiple Java Virtual Machines

Regardless of where the client and server hosts are located the Java applications running on each will require an instance of the Java Virtual Machine. (*Each computer needs the Java Runtime Environment (JRE) and any other third-party class libraries to support application execution.*) This concept is illustrated in figure 19-12.

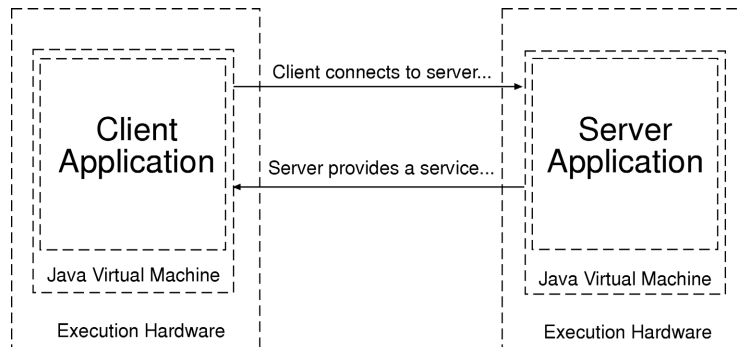


Figure 19-12: Physically Distributed Client and Server Applications Need A JVM

Quick Review

The term *application distribution* refers to where (*i.e., on what physical computer*) one or more pieces of a network application reside. Client and server applications can be deployed to the same physical computer. In this case multiple JVM instances are required to execute each client and server application. Client and server applications can be deployed to different physical computers. These computers may be in the same room or located a great distance from each other. Regardless of where the applications reside, each Java application (*client or server*) will require a JVM to execute.

Multi-Tiered Applications

Up till now I have referred to client and server applications as if they were monolithic components. In reality, modern client-server applications are logically segmented into functional layers. These layers are also referred to as *application tiers*. An application comprised of more than one tier is referred to as a *multi-tiered* application. This section discusses the concepts related to multi-tiered applications in greater detail.

Logical Application Tiers

Figure 19-13 illustrates the concept of a multi-tiered application.

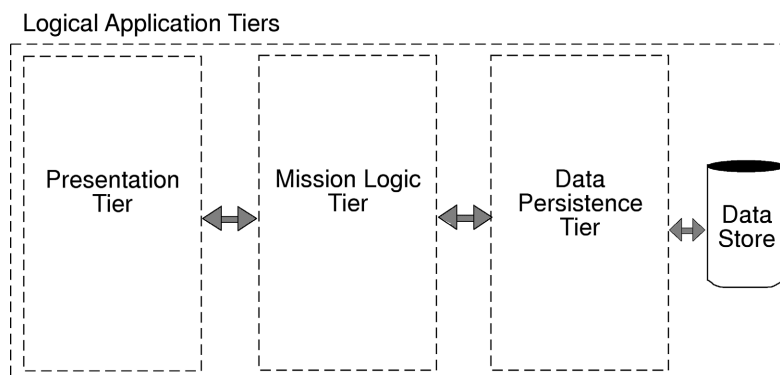


Figure 19-13: A Multi-tiered Application

In this example the application is comprised of three functional tiers: 1) Presentation Tier, 2) Mission Logic Tier, and 3) Data Persistence Tier. As their names suggest, each tier has a distinct responsibility for delivering specific application functionality. The presentation tier is concerned with rendering the user interface. The mission logic tier (*a.k.a. business logic tier*) contains the code that implements the application's services (*i.e., data processing algorithms, mission-oriented processes, etc.*) The data persistence tier is responsible for servicing the data needs (*i.e. storage and retrieval*) of the mission logic layer as quickly and reliably as possible.

Another way to think about each tier's responsibilities is as a separation of concerns:

- the presentation tier is concerned with how a user interacts with an application
- the mission logic tier is concerned with implementing mission support processes
- the data persistence tier is concerned with reliable data storage and retrieval in support of mission processes

PHYSICAL TIER DISTRIBUTION

The logical application tiers may be physically deployed on the same computer as is illustrated in figure 19-14.

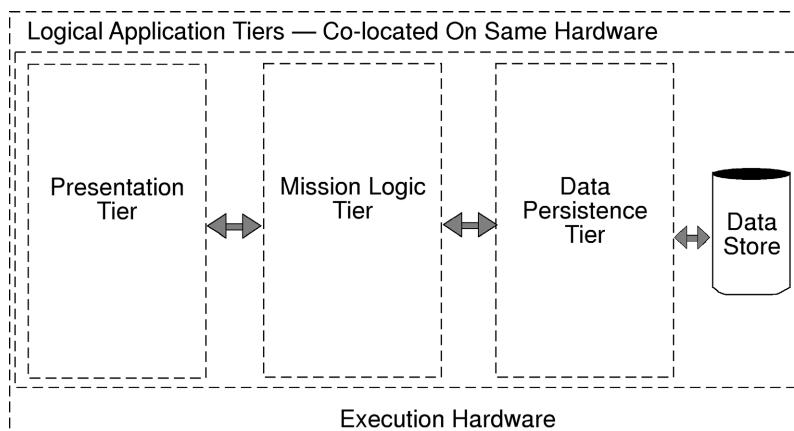


Figure 19-14: Physically Deploying Logical Application Tiers on Same Computer

It is more likely the case, however, that logical application tiers are physically deployed to separate and distinct computing nodes located great distances apart. Figure 19-15 illustrates this concept by showing each logical tier deployed to a different computer. In between this extreme lies any combination of logical tier deployments as best supports an agency's mission requirements

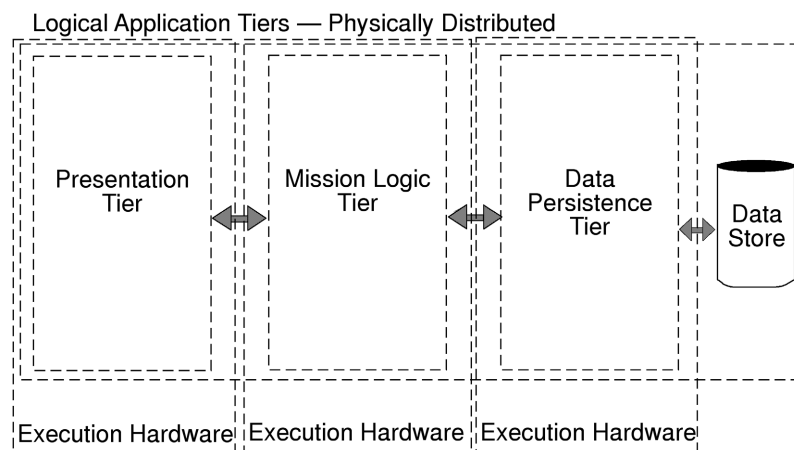


Figure 19-15: Logical Application Tiers Physically Deployed to Different Computers

Quick Review

Client and server applications can be logically separated into distinct functional areas called tiers. Applications logically segmented in this fashion are referred to as multi-tiered applications.

Three possible logical application tiers include: 1) the presentation tier, which is concerned with rendering the application's user interface, 2) the mission logic tier, which is concerned with implementing mission process logic, and 3) the data persistence tier, which is concerned with the quick and reliable delivery of data to the mission logic tier.

Multi-tiered application can be physically deployed on one computer or across several computers geographically separated by great distances.

INTERNET NETWORKING PROTOCOLS – NUTS & BOLTS

This section discusses the concepts associated with the Internet protocols and related terminology in greater detail. You'll find this background information helpful when navigating your way through the `java.net` package looking for a solution to your network programming problem.

THE INTERNET PROTOCOLS: TCP, UDP, AND IP

The Internet protocols facilitate the transmission and reception of data between participating client and server applications in a packet-switched network environment. The term packet-switched network means that data traveling along network pathways is divided into small, routable packages referred to as *packets*. If a communication link between two points on a network goes down the packets can be routed through remaining network connections to their intended destination.

The Internet protocols work together as a layered protocol stack as is shown in figure 19-16.

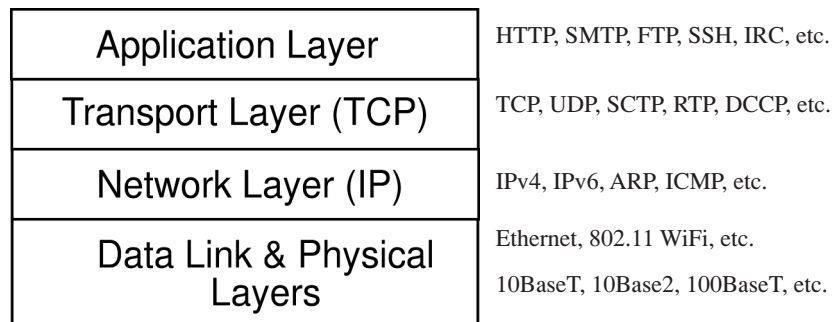


Figure 19-16: TCP/IP Protocol Stack

The layered protocol stack consists of the application layer, the transport layer, the network layer, the data link layer, and the physical layer. Each layer in the protocol stack provides a set of services to the layer above it. Several examples of protocols that may be employed at each level in the application stack are also shown in figure 19-16. For more information on protocols not discussed in this chapter please consult the sources listed in the references section.

The Application Layer

The application layer represents any internet enabled application that requires the services of the transport layer. Typical applications you may be familiar with include File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), TELNET, or a custom internet application such as one you might write. The application layer relies on services provided by the transport layer.

TRANSPORT LAYER

The purpose of the transport layer is to provide host-to-host, connection-oriented, data transmission service to the application layer. Two internet protocols that function at the transport layer include the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). Java supports both of these protocols directly in that normal network communication takes place using TCP but UDP can be utilized if required.

TRANSMISSION CONTROL PROTOCOL (TCP)

The purpose of TCP is to provide highly-reliable, host-to-host communication. To achieve this TCP manages several important issues including basic data transfer, reliability, flow control, multiplexing, connections, and precedence and security.

The sending and receiving TCP modules work together to achieve the level of service mandated by the TCP protocol. The sending TCP module packages octets of data into segments which it forwards to the network layer and the Internet Protocol (IP) for further transmission. TCP tags each octet with a sequence number. The receiving TCP module signals an acknowledgement when it receives each segment and orders the octets according to sequence number, eliminating duplicates and properly handling those that may have been received out of order.

In short — TCP guarantees data delivery and saves you the worry.

USER DATAGRAM PROTOCOL (UDP)

UDP is used to send and receive data as quickly as possible without the overhead incurred when using TCP. UDP is an extremely lightweight protocol when compared with TCP. It provides direct access to the IP datagram level. However, the quick data transmission provided by UDP comes at a price. Data is not guaranteed to arrive at its intended destination when sent via UDP.

Now, you might ask yourself, “Self, what’s UDP good for?” Generally speaking, any application that needs to send data quickly and doesn’t particularly care about lost datagrams might stand to benefit from using UDP. Examples include data streams where previously sent data is of little or no use because of its age. (*i.e.*, *stock market quote streams, voice transmissions, etc.*)

In short — UDP is faster than TCP but unreliable.

NETWORK LAYER

The network layer is responsible for the routing of data traffic between internet hosts. These hosts can be located on a local area network or on another network somewhere on the Internet. The Internet Protocol (IP) resides at this layer and provides data routing services to the transport layer protocols TCP or UDP.

INTERNET PROTOCOL (IP)

The Internet Protocol (IP) is a connectionless service that permits the exchange of data between hosts without a prior call setup. (*Hence the term connectionless.*) It packages data submitted by TCP or UDP into blocks called *datagrams*. IP uses IP addresses and routing tables to properly route datagrams to their intended destination networks.

DATA LINK & PHYSICAL LAYERS

The data link and physical layers are the lowest layers of the networking protocol stack.

THE DATA LINK LAYER

The data link layer sits below the network layer and is responsible for the transmission of data across a particular communications link. It provides for flow control and error correction of transmitted data. An example protocol that operates at the data link layer is Ethernet.

THE PHYSICAL LAYER

The physical layer is responsible for the actual transmission of data across the physical communication lines. Physical layer protocols concern themselves with the types of signals used to transmit data (*i.e.*, *electrical*, *optical*, *etc.*) and the type of media used to convey the signals (*i.e.*, *fiber optic*, *twisted pair*, *coaxial*, *etc.*).

PUTTING IT ALL TOGETHER

Computers that participate in a TCP/IP networking environment must be running an instance of the TCP/IP protocol stack as is illustrated in figure 19-17.

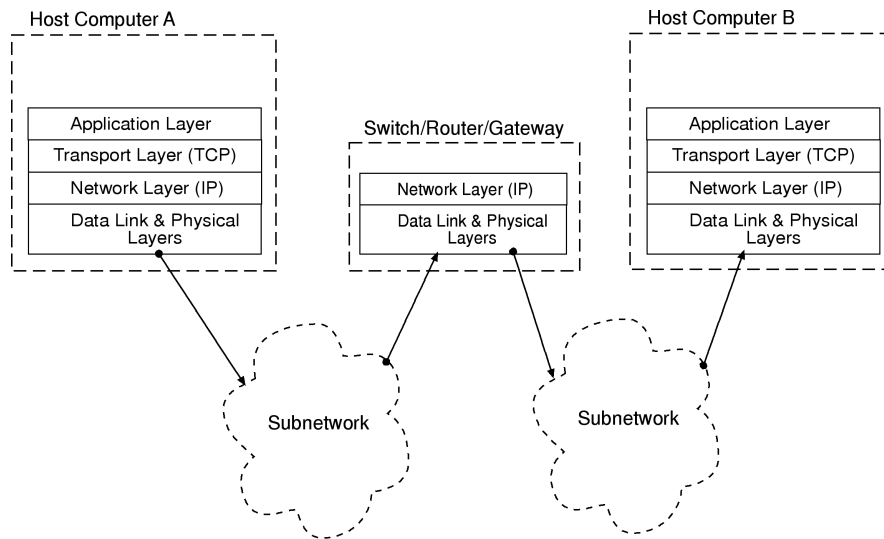


Figure 19-17: Internet Protocol Stack Operations

Referring to figure 19-17 — When host computer A sends data to host computer B via the Internet the data is passed from the application layer to the physical layer on host computer A and sent to the gateway that links the two subnetworks. At the gateway the packets are passed back up to the network layer to determine the forwarding address and repackaged and sent to the destination computer. When the packets arrive at host computer B they are passed back up the protocol stack and the original data is presented to the application layer.

WHAT YOU NEED TO KNOW

Now that you have some idea of what's involved with moving data between host computers on the Internet or on a local area network using the Internet protocols you can pretty much forget about all these nasty details. Java provides a set of classes in the `java.net` package that makes network programming easy. Java's support for network programming is discussed in the next section.

Quick Review

The Internet protocols facilitate the transmission and reception of data between participating client and server applications in a packet-switched network environment. The term packet-switched network means that data traveling along network pathways is divided into small, routable packages referred to as *packets*. If a communication link between two points on a network goes down the packets can be routed through remaining network connections to their intended destination.

The Internet protocols work together as a layered protocol stack. The layered protocol stack consists of the application layer, the transport layer, the network layer, the data link layer, and the physical layer. Each layer in the protocol stack provides a set of services to the layer above it.

The application layer represents any Internet enabled application that requires the services of the transport layer. The purpose of the transport layer is to provide host-to-host, connection-oriented, data transmission service to the application layer. Two Internet protocols that function at the transport layer include the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP guarantees data delivery and saves you the worry. UDP is faster than TCP but unreliable.

The network layer is responsible for the routing of data traffic between Internet hosts. The Internet Protocol (IP) is a connectionless service that permits the exchange of data between hosts without a prior call setup. (*Hence the term connectionless.*) It packages data submitted by TCP or UDP into blocks called *datagrams*. IP uses IP addresses and routing tables to properly route datagrams to their intended destination networks.

The data link and physical layers are the lowest layers of the networking protocol stack. The data link layer sits below the network layer and is responsible for the transmission of data across a particular communications link. The physical layer is responsible for the actual transmission of data across the physical communication lines.

Computers that participate in a TCP/IP networking environment must be running an instance of the TCP/IP protocol stack. The TCP/IP protocol stack is part of a computer's operating system.

JAVA SUPPORT FOR NETWORK PROGRAMMING

The Java platform makes network programming easy with the classes provided in the `java.net` package. This section provides a brief overview of java network programming concepts and introduces you to several frequently used classes in the `java.net` package. These classes include `ServerSocket`, `Socket`, and `URL`. You'll have a chance to apply what you learn here in chapter 20 — Client-Server Applications.

A NETWORK PROGRAMMING SCENARIO USING SOCKETS – OVERVIEW

Generally speaking, when writing a network program in Java it will be either a server application or a client application. Let's start with a server application. Figure 19-18 shows a Java server application utilizing the `ServerSocket` class to listen for incoming client connections. A `ServerSocket` object is created and bound to a particular host

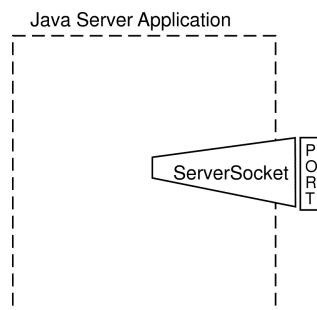


Figure 19-18: Java Server Application Utilizing a `ServerSocket` Object

IP address and port. Address and port binding can occur when the `ServerSocket` object is created. An example of a suitable address and port might be the `localhost` address (`127.0.0.1`) and port `15123`.

Client applications use the `Socket` class to establish a connection to a server application located at a specific address and port. This is illustrated in figure 19-19. When the `ServerSocket` detects an incoming client connection a server-side `Socket` is retrieved from the `ServerSocket` object and is used to communicate with the client as is shown in figure 19-20. For multi-threaded server applications the `Socket` object is passed off to a separate thread that handles communication between the server and the client. That's why it's important to understand Java threading concepts. (*See chapter 16 — Threads*)

Once the connection between the client and server is established each side can use their respective `Socket` objects to send and receive data. This is accomplished by getting the `OutputStream` and `InputStream` objects from the `Socket` object as is shown in figure 19-21. In a multi-threaded server application, after the `Socket` object is passed to a sepa-

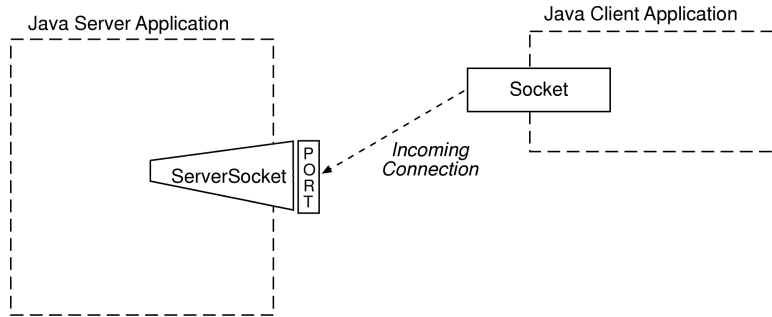


Figure 19-19: Incoming Client Connection

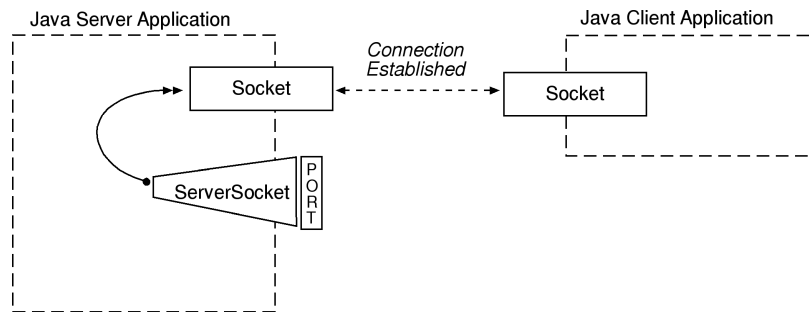


Figure 19-20: Connection Between Client & Server Established

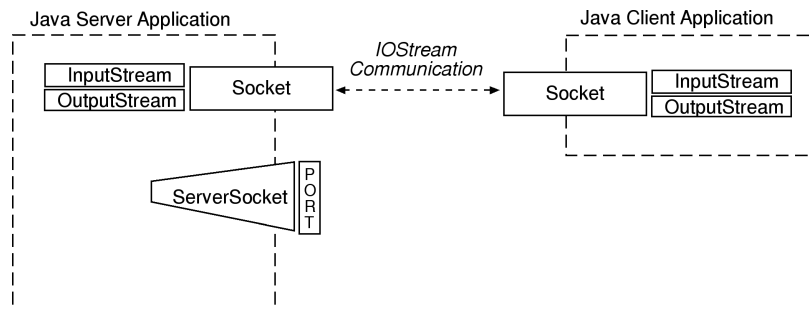


Figure 19-21: Retrieve IOStream Objects from Server and Client Socket Objects

rate thread for further processing, the ServerSocket object can return to the task of waiting for incoming client connections. In this way a multi-threaded server application can handle any number of simultaneous client connections.

THE URL CLASS

Using the ServerSocket, Socket, InputStream, and OutputStream classes is necessary if you are interested in writing client-server applications. Sometimes, however, you will simply want to access the resources provided by a network server, download the data, and manipulate it in your program. Java provides the URL and the URLConnection classes for just this purpose. This section demonstrates the use of the URL class.

The URL class represents a Uniform Resource Locator (URL) and provides methods that let you download the data referenced by the URL. The following example offers a short program that creates a URL object using the URL of a text document located on the www.warrenworks.com website and displays the text in the console.

19.1 URLTestApplication.java

```

1      import java.net.*;
2      import java.io.*;
3
4      public class URLTestApplication {
5          public static void main(String[] args){
6              try{
7                  URL url = new URL("http://www.warrenworks.com/ITP_120/SampleText.txt");
8                  DataInputStream dsr = new DataInputStream(url.openStream());
9                  String s;
10                 while((s = dsr.readLine()) != null){
11                     System.out.println(s);
12                 }
13             }catch(Exception e){
14                 e.printStackTrace();
15             }
16         } // end main
17     }

```

Referring to example 19.1 — the URL object is created on line 7 using an absolute URL reference string that represents the network resource to retrieve. On line 8 a `DataInputStream` object is created using the `URL.openStream()` method, which returns an `InputStream`, in the `DataInputStream` constructor call.

A `String` reference is declared on line 9 and used in the while loop on line 10 to store strings as they are read from the network. The results of running this program are shown in figure 19-22.

```

Terminal - tcsh
[Rick-Millers-Computer: java_book/book_code_examples/chapter_19] swodog% java URLTestApplication

*****
This is an example text document used to demonstrate the use of the Java URL class.

Ohhhh...if you love Java like I love Java...!!
*****
[Rick-Millers-Computer: java_book/book_code_examples/chapter_19] swodog%

```

Figure 19-22: Results of Running Example 19.1

Quick Review

The Java platform makes network programming easy with the classes provided in the `java.net` package. Use the `ServerSocket` and `Socket` classes when writing client-server network applications. The `ServerSocket` class is used in a server application to listen for incoming client connections. The client uses a `Socket` class to initiate connections to a server located at a particular address listening on a particular port.

When the `ServerSocket` detects an incoming client connection a server-side `Socket` object can be obtained through which the server can communicate with the client using `InputStreams` and `OutputStreams`.

The `URL` class simplifies network programming by letting you retrieve network resources via Uniform Resource Locators.

REMOTE METHOD INVOCATION (RMI) OVERVIEW

The Java platform Remote Method Invocation (RMI) mechanism provides a way for you to utilize the power and flexibility of network programming without getting your hands dirty with network programming details. (*Although your hands will still get dirty doing other things as you'll soon see!*)

RMI enables you to call a method on an object located somewhere on the network as if you were making a local method call. Figure 19-23 illustrates this concept.

Referring to figure 19-23 — a client application calls a method on a remote object as if it were an ordinary method call. Behind the scenes the Java RMI runtime environment handles all the necessary networking details such as socket set-up and object serialization. The RMI runtime on the server machine translates the incoming RMI method call request, invokes the method on the remote object, and handles the networking details for any return val-

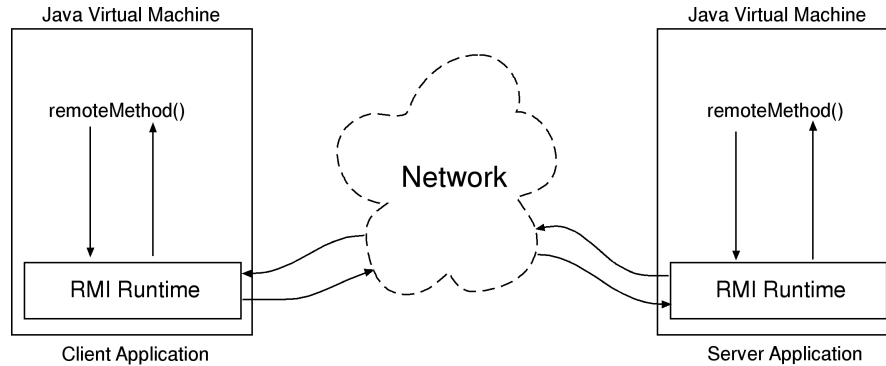


Figure 19-23: The Remote Method Invocation (RMI) Concept

ues that may result when the method call returns. The client-side RMI runtime passes the returned values to the client application, again handling all the nasty networking details such as object deserialization, etc. Again, from the client application's perspective, it appears as though it is making a local method call (*...with a few extra twists...*).

The remainder of this section steps you through the creation of a simple RMI application using Java versions 1.4.2 and 1.5 (Java 5) and the Java Remote Method Protocol version 1.2 (JRMP v1.2).

CREATING & RUNNING AN RMI APPLICATION IN JAVA 1.4.2

This section steps you through the process of creating an RMI application using Java 1.4.2. Read this section even if you are using Java 1.5 as most of what you learn here applies to creating RMI applications in Java 1.5 as well.

Required Steps

I have broken up the process of creating an RMI application into 9 sequential steps:

- Step 1: Define A Remote Interface
- Step 2: Define A Remote Interface Implementation
- Step 3: Compile The Remote Interface & Remote Interface Implementation Source Files
- Step 4: Use The `rmi.c` Tool To Generate Stub Class
- Step 5: Create Server Application
- Step 6: Create Client Application
- Step 7: Start The RMI Registry
- Step 8: Run The Server Application
- Step 9: Run The Client Application

I discuss each of these steps in detail below.

BEFORE WE GET STARTED –WHAT ARE WE GOING TO BUILD?

Before getting started, though, let's talk about what it is we are going to build. The example RMI application in this section will implement a class named `RemoteSystemMonitorImplementation` whose method named `getSystemSpecs()` can be called remotely by a network client application. The `getSystemSpecs()` method returns an array of `String` objects that represents a list of server property values such as the type of operating system, Java virtual machine version, etc.

STEP 1: DEFINE A REMOTE INTERFACE

The first step in creating an RMI application is to define an interface to the remote object. Client applications will utilize the remote object's interface to make method calls on the remote object. Example 19.2 gives the listing for the `RemoteSystemMonitorInterface.java` source file.

19.2 RemoteSystemMonitorInterface.java

```

1      import java.rmi.*;
2
3      interface RemoteSystemMonitorInterface extends Remote {
4          public String[] getSystemSpecs() throws RemoteException;
5      }

```

Referring to example 19.2 — the RemoteSystemMonitorInterface extends the Remote interface which is found in the java.rmi package. The RemoteSystemMonitorInterface declares one method named getSystemSpecs() which returns an array of Strings.

STEP 2: DEFINE A REMOTE INTERFACE IMPLEMENTATION

You must now implement the RemoteSystemMonitorInterface interface created in the previous step. Example 19.3 gives the listing for one possible approach to an implementation.

19.3 RemoteSystemMonitorImplementation.java

```

1      import java.rmi.*;
2      import java.rmi.server.*;
3
4      public class RemoteSystemMonitorImplementation extends UnicastRemoteObject
5          implements RemoteSystemMonitorInterface {
6
7          public RemoteSystemMonitorImplementation() throws RemoteException{
8              System.out.println("RemoteSystemMonitorImplementation object created!");
9          }
10
11         public String[] getSystemSpecs(){
12             String[] specs = { System.getProperty("user.name"),
13                             System.getProperty("user.home"),
14                             System.getProperty("user.dir"),
15                             System.getProperty("os.name"),
16                             System.getProperty("os.arch"),
17                             System.getProperty("java.version"),
18                             System.getProperty("java.vendor"),
19                             System.getProperty("java.vendor.url"),
20                             System.getProperty("java.vm.name"),
21                             System.getProperty("java.vm.version") };
22             return specs;
23         }
24     }

```

Referring to example 19.3 — The RemoteSystemMonitorImplementation class must extend the UnicastRemoteObject class and implement RemoteSystemMonitorInterface. The UnicastRemoteObject is found in the java.rmi.server package.

STEP 3: COMPILE THE REMOTE INTERFACE & REMOTE INTERFACE IMPLEMENTATION SOURCE FILES

When you've finished creating the implementation class compile both the interface and implementation source files. If both source files are in the same directory you can compile them using the javac compiler in the following fashion:

```
javac *.java
```

This will result in two class files named RemoteSystemMonitorInterface.class and RemoteSystemMonitorImplementation.class. The class diagram for what you have created is shown in figure 19-24.

STEP 4: USE THE rmic TOOL TO GENERATE STUB CLASS

In the same directory use the rmic tool to generate a stub class for the RemoteSystemMonitorImplementation class. Use the rmic tool in the following fashion:

```
rmic -v1.2 -d . RemoteSystemMonitorImplementation
```

The -v1.2 option tells rmic to generate only stub classes for use with Java Remote Method Protocol (JRMP) version 1.2. (JRMP v1.1 required the use of skeleton classes for use on the server. This example targets JRMP version 1.2 and skeleton classes are not necessary.)

The -d option followed by the "." tells rmic to generate the classes in the current directory. This option is the default and not necessary in this example.

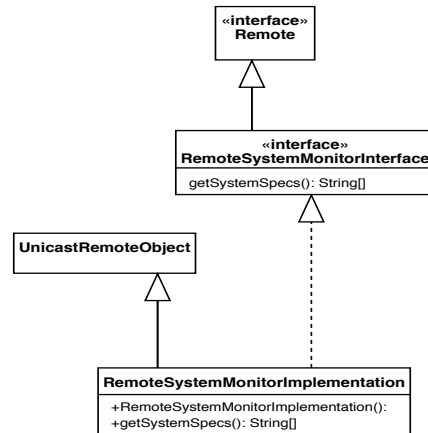


Figure 19-24: Class Diagram for RemoteSystemMonitorInterface & RemoteSystemMonitorImplementation

Running `rmic` for this example results in the creation of the `RemoteMethodMonitorImplementation_Stub.class` file in the current directory.

STEP 5: CREATE SERVER APPLICATION

Now that you have created the `RemoteSystemMonitorInterface`, `RemoteSystemMonitorImplementation`, and generated the stub class for the `RemoteSystemMonitorImplementation` (`RemoteSystemMonitorImplementation_Stub.class`) you are ready to create a server application that uses these classes. The purpose of the server application is to create an instance of `RemoteSystemMonitorImplementation` and bind the object to a service name in a running RMI registry instance. This assumes that a registry instance is running. The approach I take in this example is to programmatically create and start an instance of an RMI registry from the server application. (*This eliminates the need to execute step 7 below. However, if you write a server application that does not start a registry instance you will need to start the RMI registry externally before running a server that tries to bind a remote object to a service name with a registry.*)

Example 19.4 lists the source code for the `SystemMonitorServer` class.

19.4 `SystemMonitorServer.java`

```

1      import java.rmi.*;
2      import java.rmi.registry.*;
3
4      public class SystemMonitorServer {
5          public static void main(String[] args){
6              try{
7                  System.out.println("Creating RemoteSystemMonitorObject...");
8                  RemoteSystemMonitorInterface remote_object = new RemoteSystemMonitorImplementation();
9                  System.out.println("Starting Registry on port 1099...");
10                 LocateRegistry.createRegistry(1099);
11                 System.out.println("Registry started on port 1099");
12                 System.out.println("Binding Remote_System_Monitor service name to remote_object...");
13                 Naming.bind("Remote_System_Monitor", remote_object);
14                 System.out.println("Bind successful...");
15                 System.out.println("Ready for remote method invocations...");
16             }
17             catch(Exception e){
18                 System.out.println("Problem creating remote object!");
19                 e.printStackTrace();
20             }
21         }
22     }
  
```

Referring to example 19.4 — an instance of `RemoteSystemMonitorImplementation` is created on line 8 and assigned to the reference named `remote_object`. On line 10 the `LocateRegistry` class's `createRegistry()` method is called which starts an RMI registry instance on port 1099. (*The common RMI registry port number.*) On line 13 the object pointed to by the `remote_object` reference is bound to the service name "Remote_System_Monitor". This ser-

vice name will be used by RMI client applications to find an instance of `RemoteSystemMonitorImplementation` using the registry running on a server machine.

Compile the `SystemMonitorServer.java` source file. At this time I recommend you create separate folders or directories named `RMI_Server` and `RMI_Client`. In the `RMI_Server` folder place the following class files: `RemoteSystemMonitorInterface.class`, `RemoteSystemMonitorImplementation.class`, `RemoteSystemMonitorImplementation_Stub.class`, and `SystemMonitorServer.class`. Make a copy of the `RemoteSystemMonitorInterface.class` file and move it into the `RMI_Client` folder along with a copy of the `RemoteSystemMonitorImplementation_Stub.class` file.

STEP 6: CREATE CLIENT APPLICATION

The client application will make a remote method call on a `RemoteSystemMonitorImplementation` object located on some machine connected via the network. To do this it must get a reference to a remote object. It does this by looking it up by name on a registry running on the server. Example 19.5 gives the code for the `SystemMonitorClient` application class.

19.5 *SystemMonitorClient.java*

```

1      import java.rmi.*;
2
3      public class SystemMonitorClient {
4          public static void main(String[] args){
5              try{
6                  RemoteSystemMonitorInterface remote_system_monitor = (RemoteSystemMonitorInterface)
6                                     Naming.lookup("rmi://" + args[0] + "/Remote_System_Monitor");
7
7                  String[] specs = remote_system_monitor.getSystemSpecs();
8                  for(int i = 0; i<specs.length; i++){
9                      System.out.println(specs[i]);
10                 }
11             }catch(ArrayIndexOutOfBoundsException iae){
12                 System.out.println("Usage: java SystemMonitorClient <hostname | hostIP>");
13             }
14             catch(Exception e){
15                 e.printStackTrace();
16             }
17         }
18     }

```

Referring to example 19.5 — On line 6 a reference to a `RemoteSystemMonitorImplementation` object is fetched from a remote registry using the `Naming.lookup()` method. The `lookup()` method takes a `String` representation of a URL that represents the host location of a running RMI registry instance and the service name of the remote object to retrieve. (*Remember, in the server application the remote object instance was bound to the service name using the `Naming.bind()` method.*) The reference fetched is cast to `RemoteSystemMonitorInterface` and assigned to the reference variable named `remote_system_monitor`. The reference variable is then used to call the remote method named `getSystemSpecs()` as if it were a local method call. However, the following caveats apply: First, things might go horribly wrong when we attempt to invoke the remote method. This applies to looking up the object in the registry as well. Therefore, these method calls must be placed in the body of a `try/catch` block and the resulting exceptions handled accordingly. (*Not shown in my simple example.*) Second, the remote method call may not return right away because of network latency and the overhead incurred by going through the RMI runtime layer. However, generally speaking, so long as the network is up and running, the server is up and running, and things are generally operating normally, remote methods will invoke and return rather quickly.

When you've finished coding the client compile the source file and make sure the `RemoteSystemMonitorInterface.class` and `RemoteSystemMonitorImplementation_Stub.class` is located in the working directory or in the class-path somewhere.

You are now ready to test the RMI application.

STEP 7: START THE RMI REGISTRY

The approach I have taken in this example, which is to start the registry in the server application, renders this step unnecessary. However, if you write an RMI server application that does not programmatically start the registry then you need to issue the following command at the command prompt:

```
rmiregistry
```

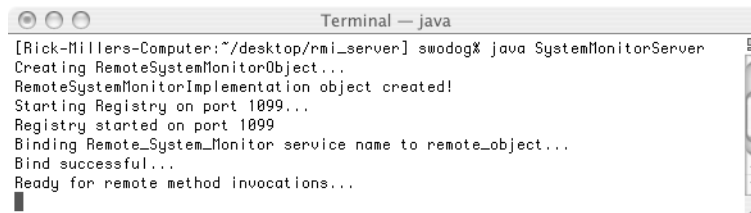
This command will start the registry service running on port 1099 which is the default RMI registry service port.

STEP 8: RUN THE SERVER APPLICATION

Start the SystemMonitorServer application on your computer of choice. You can test this RMI example on one machine if required. To start the server issue the following command at the command prompt:

```
java SystemMonitorServer
```

Figure 19-25 shows the results of running the SystemMonitorServer application.



```
Terminal — java
[Rick-Millers-Computer:~/desktop/rmi_server] swodog% java SystemMonitorServer
Creating RemoteSystemMonitorObject...
RemoteSystemMonitorImplementation object created!
Starting Registry on port 1099...
Registry started on port 1099
Binding Remote_System_Monitor service name to remote_object...
Bind successful...
Ready for remote method invocations...
```

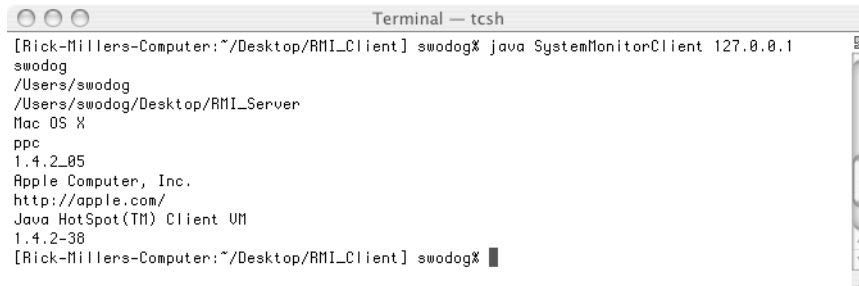
Figure 19-25: SystemMonitorServer Running on Host Machine

STEP 9: RUN THE CLIENT APPLICATION

When the SystemMonitorServer application is up and running you can run the client application. To run the client application and connect to the registry started by the SystemMonitorServer application on the local machine open a separate command or terminal window and issue the following command:

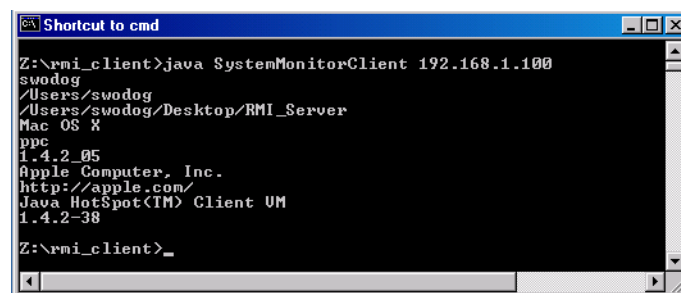
```
java SystemMonitorClient 127.0.0.1
```

The results of running the SystemMonitorClient connected to the local machine is shown in figure 19-26. Figure 19-27 shows the SystemMonitorClient application being run on a remote PC.



```
Terminal — tcsh
[Rick-Millers-Computer:~/Desktop/RMI_Client] swodog% java SystemMonitorClient 127.0.0.1
swodog
/Users/swodog
/Users/swodog/Desktop/RMI_Server
Mac OS X
ppc
1.4.2_05
Apple Computer, Inc.
http://apple.com/
Java HotSpot(TM) Client UM
1.4.2-38
[Rick-Millers-Computer:~/Desktop/RMI_Client] swodog%
```

Figure 19-26: Results of Running the SystemMonitorClient Application Connecting to the Locally Served Server Application



```
Shortcut to cmd
Z:\rmi_client>java SystemMonitorClient 192.168.1.100
swodog
/Users/swodog
/Users/swodog/Desktop/RMI_Server
Mac OS X
ppc
1.4.2_05
Apple Computer, Inc.
http://apple.com/
Java HotSpot(TM) Client UM
1.4.2-38
Z:\rmi_client>
```

Figure 19-27: Results of the SystemMonitorClient Application after Running on a remote PC.

Referring to figure 19-27 —the SystemMonitorClient can run from any machine that's also running Java and is connected to the network in such a way that it can reach the server where the registry and SystemMonitorServer is running. In this example I'm connecting from a PC to a Mac with an IP address of 192.168.1.100.

Likewise, if I run the SystemMonitorServer application on the PC I can run the SystemMonitorClient application on the Mac and invoke the remote method on the PC as figure 19-28 illustrates.



```
Terminal — tcsh
[Rick-Millers-Computer:~/Desktop/RMI_Client] swodog% java SystemMonitorClient 192.168.1.103
Rick Miller
E:\Documents and Settings\Rick Miller
E:\Documents and Settings\Rick Miller\Desktop\RMI_Server
Windows 2000
x86
1.5.0
Sun Microsystems Inc.
http://java.sun.com/
Java HotSpot(TM) Client VM
1.5.0-b64
[Rick-Millers-Computer:~/Desktop/RMI_Client] swodog% █
```

Figure 19-28: SystemMonitorClient Invoking the Remote Method on a PC Running the SystemMonitorServer Application

CREATING & RUNNING AN RMI APPLICATION IN JAVA 1.5

The steps required to create and run an RMI application using Java version 1.5 are exactly the same as those described for Java 1.4.2 with one important exception. You do not have to generate stub classes with the rmic tool if both the client and server applications are going to run in the Java 1.5 environment. The RMI runtime environment of Java 1.5 dynamically generates the stub classes for you when you invoke the remote method.

CONSIDERATIONS WHEN DEPLOYING RMI APPLICATIONS IN MIXED JVM VERSION ENVIRONMENTS

As I write this chapter Java 1.5 is available only for the Linux, Solaris, and Windows environments. The latest version of Java available for the Macintosh is version 1.4.2. (*Update: Java 1.5 for Mac OS X is here but requires an update to OS X 10.4 (Tiger)*)

In reality, you will find your Java applications being run by an assortment of JVM versions and you must consider the lowest common denominator JVM target when distributing your applications. The following guidelines will help:

SERVER JVM 1.5 - CLIENT JVM 1.5

Stub classes are not required on the client or the server side as they will be dynamically generated by the Java RMI runtime.

SERVER JVM 1.5 - CLIENT JVM 1.4.2

The lowest common denominator is JVM version 1.4.2 on the client side so you need to generate the stub classes with the rmic tool and deploy to both the client and server side.

SERVER JVM 1.4.2 - CLIENT JVM 1.5

The lowest common denominator is JVM version 1.4.2 on the server side. You'll need to generate the stub class with the rmic tool and deploy to the server side. The client side does not need the stub class.

SERVER JVM 1.4.2 - CLIENT JVM 1.4.2

Both client and server sides need the stub class.

Quick Review

Java Remote Method Invocation (RMI) provides a way for programmers to access the power and flexibility of network applications while avoiding the chore of network programming. RMI allows programmers to invoke methods on remotely located objects across a network as if they were available locally. The networking details associated with making the remote method call is handled automatically by the Java RMI runtime environment.

SUMMARY

A computer network is an interconnected collection of computing devices. Examples of computing devices include general purpose computers, special purpose computers, routers, switches, hubs, printers, personal digital assistants (*PDA*s), etc. The primary purpose of a computer network is resource sharing. A resource can be physical (*i.e. a printer*) or metaphysical (*i.e. data*).

A protocol is a specification of rules that govern the conduct of a particular activity. Entities that implement the protocol(s) for a given activity can participate in that activity. Computers participating in a computer network communicate with each other via a set of networking protocols. There are generally two types of network environments: 1) *homogeneous* - where all the computers are built by the same company and can talk to each other via that company's proprietary networking protocol, or 2) *heterogeneous* - where the computers are built by different companies, have different operating systems, and therefore different proprietary networking protocols. In today's heterogeneous computer network environment the protocols that power the Internet — Transmission Control Protocol (TCP) and Internet Protocol (IP) — collectively referred to as TCP/IP, have emerged as the standard network protocols through which different types of computers can talk to each other.

What makes the Internet so special? The answer — TCP/IP. When one computer communicates with another computer via the Internet the data it sends is separated into packets and transmitted a packet at a time to the designated computer. TCP/IP provides for packet routing and guaranteed packet delivery. Because of the functionality provided by the TCP/IP protocols the Internet is considered to be a robust and reliable way to transmit and receive data.

The terms *server* and *client* each have both a hardware and software aspect. The term *server* is often used to refer both to a piece of computing hardware on which a *server application* runs and to the server application itself. A good definition for a server then is any computer used to host one or more server applications as its primary job. A server running a server application is also referred to as a *host*. The term host extends to any computer running any application. The term client is also used to describe both hardware and software. Client hardware is any computing device that hosts an application that requires or uses the services of a server application. Client software is any application that requires or uses the services provided by a server application.

The term *application distribution* refers to where (*i.e., on what physical computer*) one or more pieces of a network application reside. Client and server applications can be deployed to the same physical computer. In this case multiple JVM instances are required to execute each client and server application. Client and server applications can be deployed to different physical computers. These computers may be in the same room or located a great distance from each other. Regardless of where the applications reside, each Java application (*client or server*) will require a JVM and supporting Java and third-party class libraries to execute.

Client and server applications can be logically separated into distinct functional areas called tiers. Applications logically segmented in this fashion are referred to as multi-tiered applications.

Three possible logical application tiers include: 1) the presentation tier, which is concerned with rendering the application's user interface, 2) the mission logic tier, which is concerned with implementing mission process logic, and 3) the data persistence tier, which is concerned with the quick and reliable delivery of data to the mission logic tier.

Multi-tiered application can be physically deployed on one computer or across several computers geographically separated by great distances.

The Internet protocols facilitate the transmission and reception of data between participating client and server applications in a packet-switched network environment. The term packet-switched network means that data traveling along network pathways is divided into small, routable packages referred to as *packets*. If a communication link between two points on a network goes down the packets can be routed through remaining network connections to their intended destination.

The Internet protocols work together as a layered protocol stack. The layered protocol stack consists of the application layer, the transport layer, the network layer, the data link layer, and the physical layer. Each layer in the protocol stack provides a set of services to the layer above it.

The application layer represents any internet enabled application that requires the services of the transport layer. The purpose of the transport layer is to provide host-to-host, connection-oriented, data transmission service to the application layer. Two internet protocols that function at the transport layer include the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP guarantees data delivery and saves you the worry. UDP is faster than TCP but unreliable.

The network layer is responsible for the routing of data traffic between internet hosts. The Internet Protocol (IP) is a connectionless service that permits the exchange of data between hosts without a prior call setup. (*Hence the term connectionless.*) It packages data submitted by TCP or UDP into blocks called *datagrams*. IP uses IP addresses and routing tables to properly route datagrams to their intended destination networks.

The data link and physical layers are the lowest layers of the networking protocol stack. The data link layer sits below the network layer and is responsible for the transmission of data across a particular communications link. The physical layer is responsible for the actual transmission of data across the physical communication lines.

Computers that participate in a TCP/IP networking environment must be running an instance of the TCP/IP protocol stack. The TCP/IP protocol stack is provided by a computer's operating system.

The Java platform makes network programming easy with the classes provided in the `java.net` package. Use the `ServerSocket` and `Socket` classes when writing client-server network applications. The `ServerSocket` class is used in a server application to listen for incoming client connections. The client uses a `Socket` class to initiate connections to a server located at a particular address listening on a particular port.

When the `ServerSocket` detects an incoming client connection a server-side `Socket` object can be obtained through which the server can communicate with the client using `InputStreams` and `OutputStreams`.

The `URL` class simplifies network programming by letting you retrieve network resources via Uniform Resource Locators.

Java Remote Method Invocation (RMI) provides a way for programmers to access the power and flexibility of network applications while avoiding the chore of network programming. RMI allows programmers to invoke methods on remotely located objects across a network as if they were available locally. The networking details associated with making the remote method call is handled automatically by the Java RMI runtime environment.

Skill-Building Exercises

1. **API Research:** Research the functionality provided by the `java.net` package. Make a list of each class and interface and write a brief description of the functionality provided by each.
2. **API Research:** Continuing to explore the `java.net` package, focus now on the `ServerSocket` and `Socket` classes. List and describe the purpose of each method provided by these classes.
3. **API Research:** Continuing to explore the `java.net` package, focus now on the `URI`, `URL`, and `URLConnection` classes. Note the purpose of each class and how they relate to each other. Note the purpose and use of each class method.
4. **Web Research:** Expand your understanding of the TCP/IP protocols. Search the web for the Internet RFCs used as references for this chapter.
5. **Web Research:** Expand your understanding of network applications. Search the web for material related to packet-switched networks, distributed applications, and multi-tiered applications.
6. **Web Research:** Expand your understanding of Java network programming. Search the web for programming examples related to the use of `ServerSocket`, `Socket`, and `URL` classes.

7. **Programming Exercise:** Write a short program that uses the URL class to access an image located somewhere on the Internet and display the image in a window.
8. **API Research:** Research the functionality provided by the java.rmi, java.rmi.activation, java.rmi.dgc, java.rmi.server, and java.rmi.registry packages.

SUGGESTED PROJECTS

1. **RMI Programming:** Given the following remote interface definition implement the interface and write the client and server applications using the steps described in this chapter. You may start the registry programmatically as I did in my example or experiment with starting it externally using the rmiregistry command.

```
1     import java.rmi.*;
2
3     interface RemoteCalculatorInterface extends Remote {
4         public double add(double a, double b) throws RemoteException;
5         public double sub(double minuend, double subtrahend) throws RemoteException;
6         public double div(double dividend, double divisor) throws RemoteException;
7         public double mul(double multiplier, double multiplicand) throws RemoteException;
8     }
```

SELF-TEST QUESTIONS

1. What is a computer network? What is the primary purpose of a computer network?
2. Describe the two types of computer networking environments.
3. Describe the purpose of the TCP/IP Internet networking protocols.
4. What's the difference between the terms server and server application? Client and client application?
5. Describe the relationship between a server application and client application.
6. List and describe at least two ways network applications can be distributed?
7. What term is used to describe a server application that can handle multiple simultaneous client connections?
8. What term is used to describe a network application logically divided into more than one functional layer? List and describe the purpose of three possible functional layers.
9. List and describe the purpose of the layers of the Internet protocol stack. Describe how data is transmitted from one computer to another via the Internet protocols.
10. What's the difference between TCP and UDP?
11. What services does IP provide?
12. Describe a client-server network application connection scenario. Focus on the purpose and use of the classes ServerSocket, Socket, and the I/O stream classes InputStream & OutputStream.
13. In what way does the URL class simplify network programming?

14. Describe in general terms how Java's Remote Method Invocation (RMI) works.

REFERENCES

Jim Farley, et. al. *Java Enterprise In A Nutshell: A Desktop Quick Reference*, Second Edition. O'Reilly & Associates, Sebastopol, CA. ISBN: 0-596-00152-5

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*, 4th Edition. O'Reilly and Associates, Inc. Sebastopol, CA. ISBN: 0-596-00283-1

The Java 2 Standard Edition Version 5 Platform API Online Documentation [<http://java.sun.com/j2se/1.5.0/docs/api/index.html>]

The Java 2 Standard Edition Version 1.4.2 Platform API Online Documentation [<http://java.sun.com/j2se/1.4.2/docs/api/index.html>]

Merlin Hughes, et. al. *Java Network Programming, Second Edition*. Manning Publications Company, Greenwich, CT. ISBN: 1-884777-49-X

William Grosso. *RMI, Dynamic Proxies, and the Evolution of Deployment*. [<http://today.java.net/pub/a/today/2004/06/01/RMI.html>]

RFC 791 - Internet Protocol

RFC 2396 - Uniform Resource Identifiers (URI): General Syntax

RFC 793 - Transmission Control Protocol

RFC 768 - User Datagram Protocol

Java RMI Release Notes for J2SE 5.0 [<http://java.sun.com/j2se/1.5.0/docs/guide/rmi/relnotes.html>]

Java RMI Release Notes for J2SE 1.4.2 [<http://java.sun.com/j2se/1.4.2/docs/guide/rmi/relnotes.html>]

Uyless Black. *Advanced Internet Technologies*. Prentice Hall Series In Advanced Communications Technologies. Prentice Hall PTR, Upper Saddle River, NJ. ISBN: 0-13-759515-8

NOTES

CHAPTER 20



Lipstick Message

CLIENT-SERVER APPLICATIONS

LEARNING OBJECTIVES

- *DEMONSTRATE YOUR ABILITY TO WRITE CLIENT-SERVER APPLICATIONS*
- *UTILIZE THE SERVERSOCKET, SOCKET, AND THREAD CLASSES TO WRITE MULTI-THREADED SERVER APPLICATIONS*
- *UTILIZE THE DATAINPUTSTREAM AND DATAOUTPUTSTREAM CLASSES TO SEND DATA BETWEEN SOCKET-BASED CLIENT AND SERVER APPLICATIONS*
- *FORMULATE AND APPLY PROPRIETARY NETWORK APPLICATION PROTOCOLS FOR USE IN CLIENT-SERVER APPLICATIONS*
- *UTILIZE REMOTE METHOD INVOCATION (RMI) TO CREATE CLIENT-SERVER APPLICATIONS*
- *UTILIZE THE PROPERTIES CLASS TO STORE AND RETRIEVE APPLICATION PROPERTIES*
- *UTILIZE THE CLASS.FORNAME() METHOD TO DYNAMICALLY LOAD CLASSES AT APPLICATION RUNTIME*
- *UTILIZE THE SINGLETON PATTERN IN AN APPLICATION*
- *UTILIZE THE FACTORY CLASS PATTERN IN AN APPLICATION*
- *ENSURE THE PROPER DESTRUCTION OF OBJECTS THROUGH THE USE OF A FINALIZE() METHOD*

INTRODUCTION

Many of the software applications you use daily are based on the client-server model. A few examples include email programs and web-based applications accessed via a web browser. Email programs and web-based applications are based upon a set of well known application protocols. (*Post Office Protocol (POP) and Simple Mail Transfer Protocol (SMTP) for e-mail programs, and Hypertext Transfer Protocol (HTTP) for web-based applications*) There are, additionally, a multitude of custom-developed client-server applications in use throughout the world providing targeted functionality to their user base. A majority of these applications utilize custom application protocols designed specifically for the task at hand. This chapter builds upon the material presented in chapter 19 and focuses on the creation of client-server applications based on sockets and Remote Method Invocation (RMI).

The discussion begins with a review of the socket-based client-server application model originally presented in chapter 19. A simple socket-based client-server application is then presented and discussed to highlight its critical features. Following this, a significant client-server application is presented that combines the socket and RMI-based client-server models. Multi-threading techniques are utilized to allow the socket-based server to simultaneously process multiple client service requests.

The secondary focus of this chapter is on the wide-array of Java platform classes and tools that enable the creation of complex client-server applications. The `DataInputStream` and `DataOutputStream` classes will be used to send and receive data between the socket-based client-server application. The `Thread` class will be used to create a multi-threaded server application, and various Swing components will be used to create user interfaces for both the client and server applications. The design techniques of inheritance and composition will be employed as well. In short, this chapter brings together a lot of the concepts discussed up to this point in the book.

A SIMPLE SOCKET-BASED CLIENT-SERVER EXAMPLE

This section presents a simple socket-based client-server application. Before getting into the code let's review the sequence of events that occur in the lifetime of a socket-based client-server application. The connection scenario starts with the client and server applications as shown in figure 20-1.

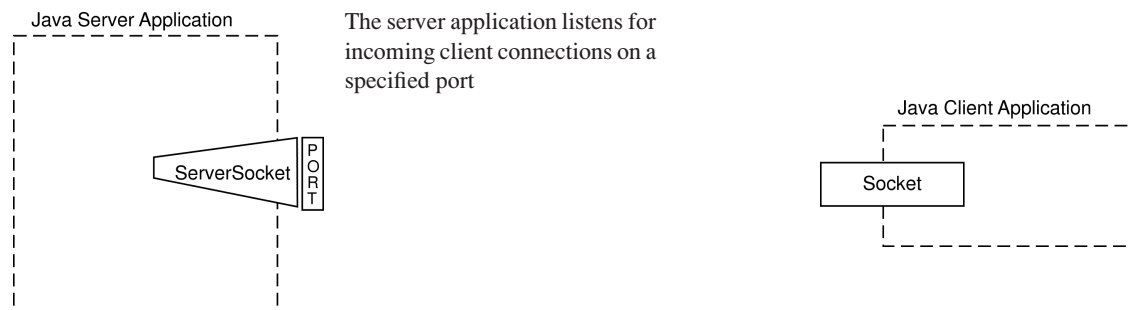


Figure 20-1: Client and Server Applications

The server application utilizes a `ServerSocket` object to listen for incoming client connections on a specified port. To connect to the server, the client application must be given the server's URL or IP address and port number. The incoming client connection is illustrated in figure 20-2. When the incoming client connection is detected a `Socket` object is retrieved from the `ServerSocket` object as is shown in figure 20-3.

Once the connection is successfully established and the `Socket` objects are set up on both ends of the connection the `Socket` objects are used to retrieve the `IOStream` objects through which communication will take place. This concept is illustrated in figure 20-4. The `IOStream` objects (*an `InputStream` and `OutputStream`*) are used to create the required `IOStream` object suitable to the communication task at hand. For instance, if you intend to send serialized objects over the wire then you need to create an `ObjectInputStream` and an `ObjectOutputStream`. If you want to send Strings and primitive types then perhaps the `DataInputStream` and `DataOutputStream` classes will suffice.

In multi-threaded server applications the client communication activities are handled in a separate thread.

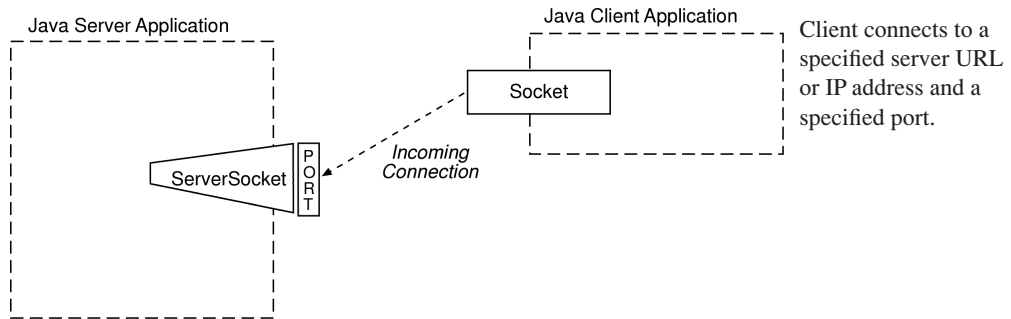


Figure 20-2: Incoming Client Connection

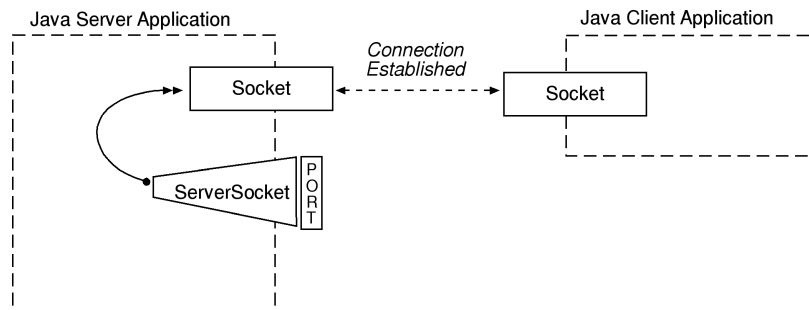


Figure 20-3: The Connection is Established — There are Sockets at Both Ends of the Connection

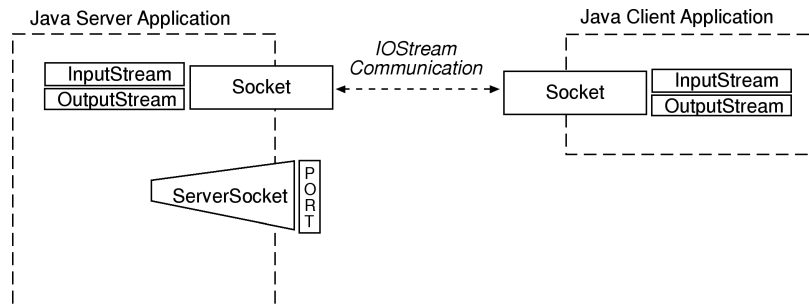


Figure 20-4: The Socket Objects are Used to Retrieve the IOStream Objects

When the client disconnects, the socket must be closed with a call to the `Socket.close()` method.

Now that we've reviewed the basic activities associated with a socket-based client-server application let's take a look at some code that implements a simple socket-based client-server application. I'll start with the server application.

Simple SERVER Application

Example 20.1 gives the code for an application class named `SimpleServer`. The `SimpleServer` application implements a simple socket-based single-threaded server application. It's simple because all it does is echo message strings back to a client application. It's single threaded because it can only process one client communication session at a time.

```

1      import java.net.*;
2      import java.io.*;
3
4      public class SimpleServer {
5          public static void main(String[] args){
6              try{
7                  System.out.println("Creating ServerSocket object binding to port 5001...");
8                  ServerSocket server_socket = new ServerSocket(5001);
9                  boolean keep_going = true;

```

20.1 *SimpleServer.java*

```

10     while(keep_going) {
11         System.out.println("Listening for incoming client connections on port 5001...");
12         Socket socket = server_socket.accept();
13         System.out.println("Incoming client connection detected...");
14         System.out.println("Creating Input- and OutputStream objects...");
15
16         DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
17         DataInputStream dis = new DataInputStream(socket.getInputStream());
18
19         String input_string = "Ready";
20         System.out.println("Processing client String input...");
21         try{
22             while((input_string = dis.readUTF()) != null){
23                 System.out.println(input_string);
24                 dos.writeUTF("Echo from server: " + input_string);
25
26                 if(input_string.equals("disconnect client")){
27                     System.out.println("Client disconnected...");
28                     socket.close();
29                     break;
30                 }else if(input_string.equals("shutdown server")){
31                     socket.close();
32                     keep_going = false;
33                     break;
34                 }
35             } // end inner while loop
36         }catch(EOFException ignored){ }
37         catch(Exception e1){
38             e1.printStackTrace();
39         }
40     } // end outer while loop
41     System.out.println("Shutting down server...");
42 }catch(Exception e){
43     e.printStackTrace();
44 }
45 } // end main()
46 } // end SimpleServer class definition

```

Referring to example 20.1 — when the SimpleServer application starts up it creates a ServerSocket object on line 8 that is bound implicitly to the default host IP address and explicitly to port 5001. The rest of the application is then enclosed in a while loop that repeatedly executes a few basic server tasks. First, utilizing the ServerSocket object on line 12, it attempts to create a Socket object by calling the ServerSocket.accept() method. The accept() method blocks right there until it detects an incoming client connection. When an incoming client connection is detected the accept() method returns and the Socket object is created. Next, the DataOutputStream and DataInputStream objects are created by calling the Socket.getOutputStream() and Socket.getInputStream() methods respectively. Once the IOStream objects are created the server application is ready to communicate with the client.

Client communication is handled in the inner while loop that begins on line 22. The DataInputStream object is used to read a UTF string from the client using the readUTF() method. (*The client must send a UTF string!*) The string sent by the client is then written to the server console and immediately sent back to the client application.

Examine line 26 — if the input_string equals “disconnect client” the socket is closed and the inner while loop is exited with the break statement. The server application then returns to listening for incoming client connections and the process repeats when another client connection is detected. If, however, the input_string equals “shutdown server” then the socket is closed and the keep_going metaphor value is set to false and the outer while loop exits. This causes the SimpleServer application to exit.

Most of the code for the SimpleServer application is enclosed in a try-catch block. When dealing with network programming issues and IOStream communication a multitude of exceptions must be addressed and properly handled. One exception that is specifically addressed in the SimpleServer example is the EOFException that will result if a client connection is suddenly dropped. The EOFException is thrown by the DataInputStream.readUTF() method. (*In addition to IOException and UTFDataFormatException*). Otherwise, the SimpleServer application simply tries to recover somewhat gracefully from the myriad exceptions the can possibly be thrown during the several stages of the server life cycle. A robust server application must do everything in its power to properly handle exceptional conditions.

Now that you’ve seen the SimpleServer application code let’s move on to see the code for a simple client application that can connect to a running instance of SimpleServer and exchange messages with it.

Simple Client Application

Example 20.2 provides the source code for an application class named `SimpleClient`. The `SimpleClient` application provides a basic GUI front-end to a client application that can connect to and exchange messages with a running instance of `SimpleServer`. Read through the code and at the end I'll discuss some of its important features.

20.2 SimpleClient.java

```

1      import java.net.*;
2      import java.io.*;
3      import javax.swing.*;
4      import java.awt.event.*;
5      import java.awt.*;
6
7      public class SimpleClient extends JFrame implements ActionListener {
8
9          private JPanel panel1 = null;
10         private JPanel panel2 = null;
11         private JTextArea textareal = null;
12         private JTextArea textarea2 = null;
13         private JButton button1 = null;
14         private JScrollPane scrollpanel1 = null;
15         private JScrollPane scrollpane2 = null;
16
17         Socket socket = null;
18         DataInputStream dis = null;
19         DataOutputStream dos = null;
20
21         public SimpleClient(String url){
22             super("SimpleClient");
23             panel1 = new JPanel();
24             panel2 = new JPanel();
25             textareal = new JTextArea(12, 25);
26             textareal.append("Enter your message here.");
27             scrollpanel1 = new JScrollPane(textareal);
28             textarea2 = new JTextArea(12, 25);
29             textarea2.setEnabled(false);
30             textarea2.append("Server response messages will appear here.\n");
31             scrollpane2 = new JScrollPane(textarea2);
32             panel1.add(scrollpanel1);
33             panel1.add(scrollpane2);
34             button1 = new JButton("Send");
35             button1.addActionListener(this);
36             panel2.add(button1);
37             this.getContentPane().add(panel1);
38             this.getContentPane().add(BorderLayout.SOUTH, panel2);
39             this.setSize(620, 300);
40             this.setLocation(200, 200);
41             this.show();
42
43             try{
44                 System.out.println("Creating Socket object...");
45                 socket = new Socket(url, 5001);
46                 System.out.println("Creating IOStream objects...");
47
48                 dos = new DataOutputStream(socket.getOutputStream());
49                 dis = new DataInputStream(socket.getInputStream());
50
51             }catch(Exception e){
52                 e.printStackTrace();
53             }
54             System.out.println("Ready to send commands to the server...");
55         }
56
57         public void actionPerformed(ActionEvent ae){
58             if(ae.getActionCommand().equals("Send")){
59                 System.out.println(textareal.getText());
60
61                 System.out.println("Sending text to server...");
62                 try{
63                     dos.writeUTF(textareal.getText());
64                 }catch(Exception e){
65                     e.printStackTrace();
66                 }
67
68                 System.out.println("Reading text from server...");
69                 try{
70                     textarea2.append(dis.readUTF() + "\n");
71                 }catch(Exception e){
72                     e.printStackTrace();

```



```

73         }
74
75         if(textarea1.getText().equals("disconnect client") ||
76             textarea1.getText().equals("shutdown server")){
77             System.out.println("Client shutting down...");
78             try{
79                 socket.close();
80             }catch(Exception e){
81                 e.printStackTrace();
82             }
83             System.exit(0);
84         }
85     }
86 } // end actionPerformed() method
87
88
89 public static void main(String[] args){
90     System.out.println("SimpleClient lives!");
91     try{
92         new SimpleClient(args[0]);
93     }catch(NullPointerException npe){
94         System.out.println("Usage: java SimpleClient <host>");
95     }
96     catch(ArrayIndexOutOfBoundsException oobe){
97         System.out.println("Usage: java SimpleClient <host>");
98     }
99     catch(Exception e){
100        e.printStackTrace();
101    }
102 } // end main()
103 }

```

Referring to example 20.2 — on lines 9 through 19 the `SimpleClient` class declares several Swing component fields, a `Socket` field, a `DataInputStream` field, and a `DataOutputStream` field. The constructor method initializes the Swing components as expected. Starting on line 43 the `Socket` and `IOStream` objects are created inside the body of a try-catch block. Notice that the `Socket` object is created first. If this is successful then the `IOStream` objects are created by calling the `Socket.getInputStream()` and `getOutputStream()` methods as was done in the server code.

Once the application is up and running all interaction between the client and server application takes place in the body of the `actionPerformed()` method that starts on line 57. Messages entered in `textarea1` are sent to the server. Messages received from the server are written to `textarea2`. The message “disconnect client” causes the client to shut-down and the server to return to listening for incoming client connections. The message “shutdown server” shuts down both the client and server applications.

The `main()` method begins on line 89. The `SimpleClient` application must be started with a host URL or IP address. If the user fails to enter a URL or IP address the application will print a short message showing how to properly use the application.

Now, let’s run these applications and see how they work.

RUNNING THE EXAMPLES

Start by running the `SimpleServer` application first. It must be running before it can service incoming client connections. Figure 20-5 shows how the `SimpleServer` console looks after it has started and is waiting for incoming client connections. You can now start up an instance of `SimpleClient` and connect to the server.



Figure 20-5: `SimpleServer` Running & Waiting for Incoming Client Connections

Figure 20-6 shows the console and GUI for the `SimpleClient` application. The `SimpleClient` application is started with the IP address of 127.0.0.1 (*the localhost address*) since this is where the `SimpleServer` application is running. If it were running on another machine on your network you would need to use the IP address for that machine.

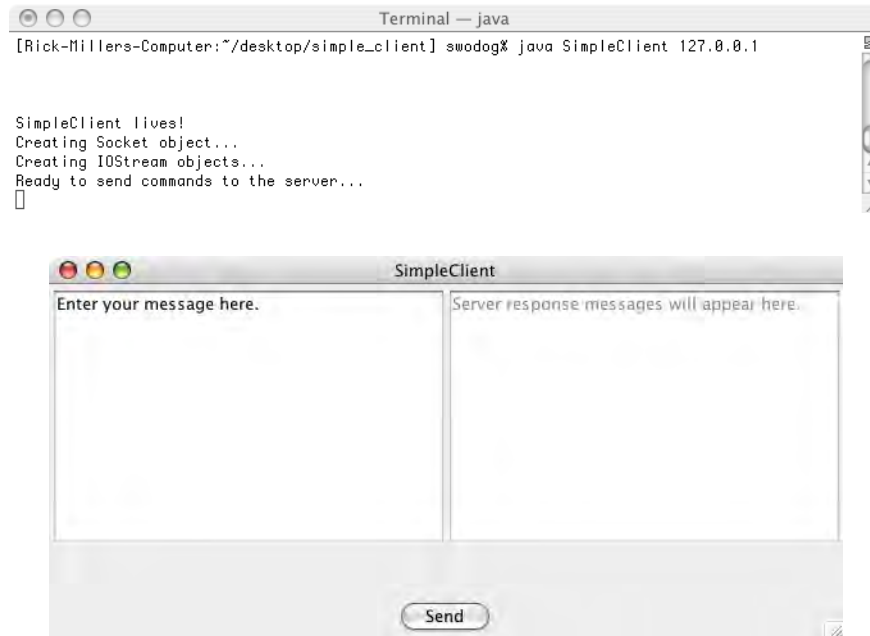


Figure 20-6: SimpleClient Console Output and GUI

When the SimpleClient application starts up it prints out a few diagnostic messages as it creates the Socket object and IOStream objects. It then prints a message to the console indicating its readiness to send messages to the server.

When the client connection is made the SimpleServer application prints diagnostic messages to the console indicating that it is creating the IOStream objects and processing client string input. Figure 20-7 shows the SimpleServer console at this stage of the game.

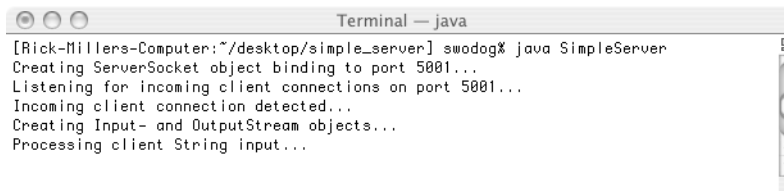


Figure 20-7: SimpleServer Console After Detecting Incoming Client Connection

To send a message to the server using the SimpleClient application enter a message in the left text area. Note that all the text contained in the left text area will be sent to the server and echoed back from the server into the right text area. Figure 20-8 shows an exchange of several messages between the client and server applications.

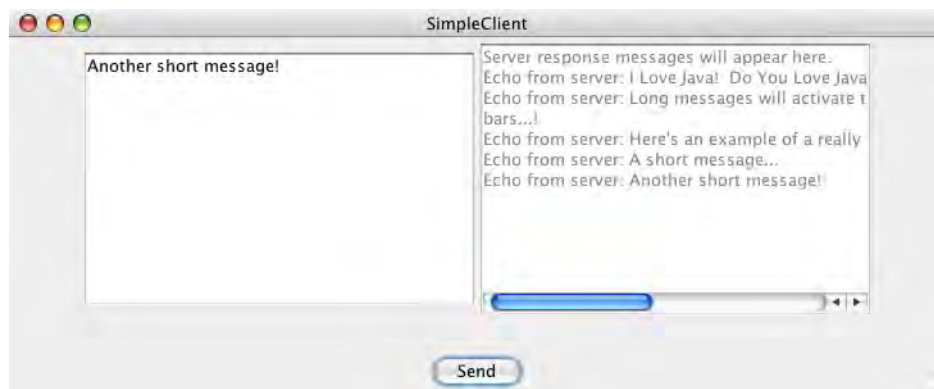


Figure 20-8: Several Messages Exchanged with the Server From SimpleClient

To shut down the client enter “disconnect client” in the left text area and click the send button. To shut down both the client and the server applications send the “shutdown server” message.

A FEW WORDS ABOUT PROTOCOL

The SimpleServer application establishes a set of ground rules for clients to follow when attempting to communicate. In the case of SimpleServer the protocol is simple: Send it a UTF message and it will send it right back to you, tit-for-tat. The strings “disconnect client” and “shutdown server” can be considered special commands the server recognizes to execute two special functions. (*returning to listen for incoming client connections and shutting down*) Clients that attempt to communicate with SimpleServer must be clued in to its protocol requirements, otherwise, not much will happen when the connection is made, if it’s made at all!

Another aspect of the SimpleServer protocol is that it is *connection oriented*. This means that the client and server applications maintain a socket connection open for the duration of their communication session.

Quick Review

Simple socket-based server applications utilize a ServerSocket object to listen for incoming client connections. A ServerSocket object listens for incoming connections on a specified port. Client applications utilize a Socket object to connect to a server application on a specified URL or IP address and port number. The localhost IP address can be used during testing.

The Socket object is used to create the InputStream objects using the `getInputStream()` and `getOutputStream()` methods. Once the InputStream objects are created on both the client and server ends the applications can communicate with each other via an established set of rules. (*a.k.a., a custom application protocol*). A client-server protocol is connection oriented if a socket connection is maintained open between the client and server applications for the duration of their communication session.

PROJECT SPECIFICATION

The remainder of this chapter will focus on the construction of a multi-threaded client-server application that satisfies the requirements as set forth in the project specification given in table 20-1.

Project Specification Client-Server Application

Objectives:

- Utilize the ServerSocket, Socket, DataInputStream, and DataOutputStream classes in a client-server application.
- Utilize the Thread class to create a multi-threaded server application.
- Demonstrate your ability to create and implement custom application protocols.
- Demonstrate your ability to translate custom application protocols into actions executed by the server application.
- Utilize Remote Method Invocation (RMI) to implement a client-server application.

Tasks:

- General Description: Write a client-server application that allows you to remotely control the movements of a robot rat around a floor. A user must be able to connect to the server via a client application and control the movements of a robot rat that is associated with that client session. Give users the ability to move their robot rat in the following directions: N, NE, E, SE, S, SW, W, and NW.

Table 20-1: Client-Server Project Specification

- Server Specification: The server application must be capable of handling socket-based and RMI-based client connections simultaneously. The socket-based portion of the server application must process client communications in a separate thread. The server must display an image of a user's robot rat upon the floor and update the location of the robot rat image as the user moves it about the floor.

- Client Specification: Users interact with the robot rat server via a client application. You must implement two client applications: 1. an RMI-based client, and 2) a socket-based client. You may optionally combine the two client applications if you desire. Give the client application a graphical user interface utilizing Swing components. The user interface might take the form of a 3 x 3 grid of JButtons that allows users to easily send directional move commands to their robot rats. You must also allow users to specify the URL or IP address of the robot rat server when they start up the client application.

- Resources: You may use the following image for the robot rats if you desire:



Figure 20-9: rat.gif

The rat.gif image can be found on the companion CD in the NetRatServer project folder.

- Hints: Attack this project according to the project approach strategy. Carefully analyze the project specification and formulate a rough design and implementation approach. Implement the project in stages, starting with a subset of the server application functionality. Continue the design-build-test cycle until you have completed the project.

Look at the Canvas class as a foundation for the robot rat floor. Implement the rat as a stand-alone class that contains its image and its position upon the floor.

Table 20-1: Client-Server Project Specification

As you can tell from reading the project specification this is no small project. The server application must provide the capability to process service requests from socket-based and RMI-based client applications simultaneously. The socket-based portion of the server code must be multi-threaded. This means that when the server detects an incoming socket-based client connection it must pass off that connection to a separate thread for processing. Given the complexity of this application it will be helpful to spend time discussing in greater detail some of the issues surrounding its possible design.

CLIENT-SERVER PROJECT INITIAL DESIGN CONSIDERATIONS

There are quite a few complex issues that must be worked out, at both high and low levels, in order to implement a suitable solution to this problem. Before writing one line of code you must have, at a minimum, a good idea of the direction you are heading design-wise.

NOUN-VERB ANALYSIS

A good place to start is by doing a basic noun-verb analysis of the project specification. The reason for completing such an analysis is to get your creative juices flowing. One good design idea can potentially yield many others. Table 20-2 offers a first attempt at such an analysis.

Nouns	Verbs
robot rat rat floor client server socket-based client RMI-based client protocol thread connections client application server application user's robot rat rat's position	move move north move south move east move west move north-west move north-east move south-east move south-west connect

Table 20-2: Client-Server Project Noun-Verb Analysis

High-Level Application Operations

The list of nouns and verbs, although not exhaustive, is complete enough to allow you to march forward. At this point you know that the client and server applications will communicate with each other via sockets and RMI. The client application will send commands to the server application via the network. There will be two types of clients: socket-based and RMI-based. The server will receive the client commands and translate them into robot-rat movements upon the floor.

Armed with this high-level application operational description and the noun-verb lists it will now prove helpful to pick several nouns from the list that appear to be good candidates for classes and assign to them some responsibilities or make comments about their functionality. During this analysis you may discover other derived nouns and verbs (*derived requirements*) that can be included in the analysis as well. Table 20-3 makes a first attempt at class responsibility assignment.

Noun	Possible Class Name	Responsibilities & Design Comments
rat	Rat	Embodies a single rat object. Contains a reference to a rat image (<i>rat.gif</i> or <i>other developer-provided image</i>) and its position upon the floor. The position could be a separate class but in this example the Rat class will contain x and y coordinates indicating its position upon the floor. There is one Rat object for each client session. This means there could potentially be many Rat objects on the floor at any given time. Clearly there must be a <u>collection of Rats</u> somewhere.
collection of Rats	java.util.Vector	An instance of the Vector class will be used to hold a collection of references to Rat objects. When a new user connects to the server application their Rat will be added to the Vector. Maybe the Floor class can iterate through the Vector of Rats and draw each Rat's image on itself after each move.
robot rat	RobotRat	We need a class that can control the movements of individual rats. The responsibilities of the RobotRat class will include making rats turn in different directions and move about the floor. There will be a one-to-one mapping between a Rat object and its controlling RobotRat object. The RobotRat will need access to the collection of rats so it can insert its rat into the collection. The RobotRat class might also be a good candidate to implement the server-side RMI functionality.

Table 20-3: Class Responsibility Assignment

Noun	Possible Class Name	Responsibilities & Design Comments
floor	Floor	A Floor is a graphical representation of the robot rat movement area. The Floor class will extend the java.awt.Canvas class as per the hint given in the project specification. How will the rat images get drawn to the floor? How will a rat image be updated on the floor after a user moves a rat? What does it mean to move a rat? Also, there will be only one floor object upon which all user's rats are moved.
Thread	ThreadedClientProcessor	The ThreadedClientProcessor class will be responsible for managing socket-based communication between the client and server applications. An instance of the ThreadedClientProcessor class will be created when the server detects a new incoming socket-based client connection. The Socket object retrieved from the server application's ServerSocket object will be passed to a ThreadedClientProcessor object for client session processing.
server application	NetRatServer	The NetRatServer is the main application class. Its responsibilities include overall server application initialization. When you start-up the server you are running an instance of NetRatServer. The NetRatServer must manage both the RMI- and Socket-based server application aspects.
RMI-based client	RMI_NetRatClient	The RMI_NetRatClient class will implement the RMI client application. The RMI_NetRatClient class will display a GUI interface which will contain buttons that can be used to send commands to the NetRatServer application.
socket-based client	Socket_NetRatClient	The Socket_NetRatClient class will implement the socket-based client application. The Socket_NetRatClient class will display a GUI interface which will contain buttons that can be used to send commands to the NetRatServer application.

Table 20-3: Class Responsibility Assignment

Table 20-3 offers a good start at assigning initial class responsibilities. There are still a few details to be worked out but there is enough information available to proceed with an initial design. Figure 20-10 offers a first draft class diagram that highlights the important relationships between the server-side classes.

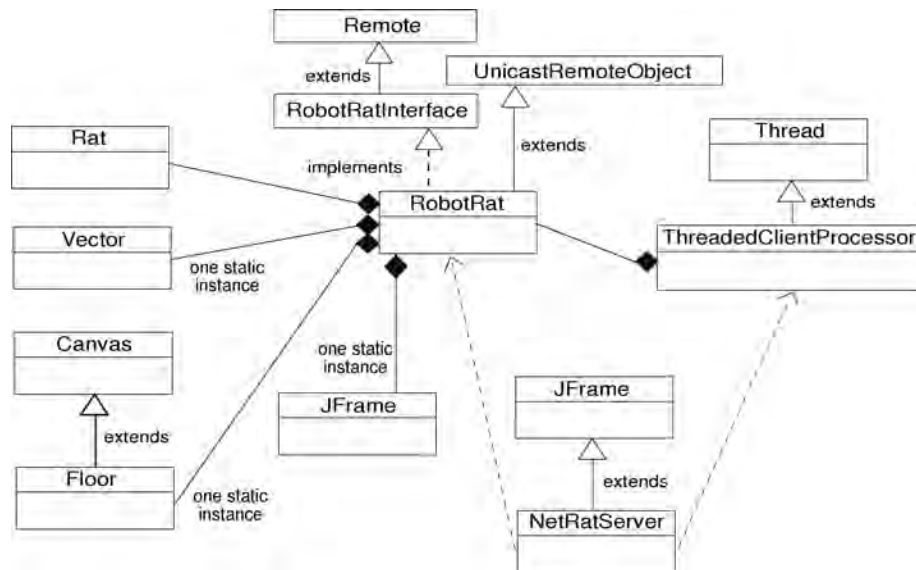


Figure 20-10: First Draft Class Diagram for the NetRatServer Application

Referring to figure 20-10 — The RobotRat class will contain one static instance of a Vector and a Floor. The Floor will need to be drawn into a Frame so RobotRat will also contain a static instance of a JFrame. These three objects will be shared by all instances of RobotRat. Each RobotRat instance will manage its own private instance of Rat. The RobotRat class will also serve as the class that implements the server-side RMI functionality.

The ThreadedClientProcessor class will contain an instance of RobotRat and will translate socket-based client commands into RobotRat method calls.

The NetRatServer class will serve as the main application. It will start the RMI registry and bind an instance of RobotRat with its service name. The NetRatServer class will also listen for incoming socket-based client connections and pass the socket off to an instance of ThreadedClientProcessor for further processing.

For initial testing purposes the NetRatServer application can display a window with buttons to send test commands to robot rats. That's why figure 20-10 shows NetRatServer extending JFrame.

HOW TO PROCEED

With an overall design approach in hand and a good sense of direction it's time to start coding. As you begin to implement the server application some aspects of the design will evolve as your understanding of the problem deepens. As usual, a good way to proceed with the implementation effort is to select a subset of the design, code it up, test it, reflect on the design up to that point, make changes where necessary, then repeat the process with another subset of the design.

QUICK REVIEW

Proceed first with a client-server project by giving sufficient analysis to the project specification. Use noun-verb analysis to discover candidate application components and actions. Sketch out a rough application design to use as an implementation road map. Make a list of candidate application classes and assign to them an initial set of responsibilities. The objective of the analysis and design phase is to provide a starting point for implementation and get the creative juices flowing.

Socket-based client-server applications will utilize classes found in the java.net package. RMI-based client-server applications will utilize classes found in the java.rmi package and its related packages. However, to fully implement a client-server application requires the utilization of many different Java platform classes.

IMPLEMENTING NETRATSERVER – FIRST ITERATION

Table 20-4 lists the design considerations and design decisions for the first iteration.

Check-Off	Design Considerations	Design Decisions
	Objectives for the first iteration	Implement the Rat, Floor, RobotRat, and NetRatServer classes concentrating on drawing a collection of Rat images on the Floor and testing their move capability. Network issues will be ignored for now.
	Rat class	The Rat class will keep track of its x and y coordinates upon the floor. The Rat class will also have an associated image that will get painted to the floor.
	Floor class	The Floor class will reference a Vector of Rat objects and paint each Rat's image on the Floor using its x and y coordinates.
	java.util.Vector class	The Vector class will be used to maintain a collection of Rat objects.
	javax.JFrame	The Floor will be drawn into a static instance of JFrame.

Table 20-4: First Iteration Design Considerations and Decisions

Check-Off	Design Considerations	Design Decisions
	RobotRat class	The RobotRat class will contain one static instance of Floor, Vector, and JFrame. These static instances will be used by all instances of RobotRat. An instance of RobotRat will maintain and move its own private instance of a Rat.
	RMI Functionality	Ignored in this iteration.
	NetRatServer	The NetRatServer class will be used to test the functionality of the classes developed during this iteration.

Table 20-4: First Iteration Design Considerations and Decisions

FIRST ITERATION CODE

The following code is developed for this first iteration:

20.3 Rat.java

```

1      import java.awt.*;
2      import javax.swing.*;
3
4      public class Rat {
5          private Image _its_image = null;
6          private int _x = 0;
7          private int _y = 0;
8
9          public Rat(String image_name) {
10             Toolkit tool_kit = Toolkit.getDefaultToolkit();
11             try {
12                 _its_image = tool_kit.getImage(image_name);
13             } catch (Exception e) {
14                 System.out.println("Rat constructor: Problem loading or creating Rat image!");
15                 e.printStackTrace();
16             }
17             _x = 0;
18             _y = 0;
19         }
20
21         public Rat() {
22             this("rat.gif");
23         }
24
25         public Image getImage() {
26             return _its_image;
27         }
28
29         public int getX() {
30             return _x;
31         }
32
33         public int getY() {
34             return _y;
35         }
36
37         public void setX(int x) {
38             _x = x;
39         }
40
41         public void setY(int y) {
42             _y = y;
43         }
44     } // end Rat class definition

```

20.4 Floor.java

```

1      import java.awt.*;
2      import java.util.*;
3
4      public class Floor extends Canvas {
5          private Vector _rats = null;
6

```



```

7     public Floor(Vector rats){
8         _rats = rats;
9         this.setBackground(Color.BLACK);
10    }
11
12    public void paint(Graphics g){
13        for(int i = 0; i<_rats.size(); i++){
14            if(_rats.elementAt(i) != null){
15                g.drawImage(((Rat)_rats.elementAt(i)).getImage(),((Rat) _rats.elementAt(i)).getX(),
16                    ((Rat) _rats.elementAt(i)).getY(), this);
17            }
18        }
19    }
20 } // end Floor class definition

```

20.5 RobotRat.java

```

1     import javax.swing.*;
2     import java.util.*;
3
4     public class RobotRat {
5
6         private static JFrame _frame = null;
7         private static Floor _floor = null;
8         private static Vector _rats = null;
9         private Rat _its_rat = null;
10
11        static {
12            _rats = new Vector();
13            _floor = new Floor(_rats);
14            _frame = new JFrame("Rat Movement Floor");
15            _frame.getContentPane().add(_floor);
16            _frame.setSize(300, 300);
17            _frame.setLocation(200, 200);
18            _frame.show();
19        }
20
21
22        public RobotRat() {
23            _its_rat = new Rat();
24            _rats.addElement(_its_rat);
25            _floor.repaint();
26        }
27
28        public void moveEast(){
29            _its_rat.setX(_its_rat.getX() + 3);
30            _floor.repaint();
31        }
32
33        public void moveSouth(){
34            _its_rat.setY(_its_rat.getY() + 3);
35            _floor.repaint();
36        }
37
38        public void moveWest(){
39            _its_rat.setX(_its_rat.getX() - 3);
40            _floor.repaint();
41        }
42
43        public void moveNorth(){
44            _its_rat.setY(_its_rat.getY() - 3);
45            _floor.repaint();
46        }
47
48        public void moveNorthWest(){
49            _its_rat.setY(_its_rat.getY() - 3);
50            _its_rat.setX(_its_rat.getX() - 3);
51            _floor.repaint();
52        }
53
54        public void moveSouthWest(){
55            _its_rat.setY(_its_rat.getY() + 3);
56            _its_rat.setX(_its_rat.getX() - 3);
57            _floor.repaint();
58        }
59
60        public void moveNorthEast(){
61            _its_rat.setY(_its_rat.getY() - 3);
62            _its_rat.setX(_its_rat.getX() + 3);
63            _floor.repaint();
64        }

```

```

65
66     public void moveSouthEast(){
67         _its_rat.setY(_its_rat.getY() + 3);
68         _its_rat.setX(_its_rat.getX() + 3);
69         _floor.repaint();
70     }
71 } // end RobotRat class definition

                                                                    20.6 NetRatServer.java

1     import javax.swing.*;
2     import java.awt.*;
3     import java.awt.event.*;
4
5     public class NetRatServer extends JFrame implements ActionListener {
6
7         private JButton button1 = null;
8         private RobotRat _r1 = null;
9         private RobotRat _r2 = null;
10
11     public NetRatServer(){
12
13         _r1 = new RobotRat();
14         _r2 = new RobotRat();
15         button1 = new JButton("move");
16         button1.addActionListener(this);
17         this.getContentPane().add(button1);
18         this.setSize(100, 100);
19         this.setLocation(300, 300);
20         this.show();
21     }
22
23     public void actionPerformed(ActionEvent ae){
24         if(ae.getActionCommand().equals("move")){
25             _r1.moveEast();
26             _r2.moveSouthEast();
27         }
28     }
29
30     public static void main(String[] args){
31         System.out.println("NetRatServer Lives!!");
32         new NetRatServer();
33     } // end main()
34 } // end NetRatServer class definition

```

Referring to example 20.3 through 20.6 — The Rat class implementation is fairly straight forward. It has three private fields representing its x and y coordinates and its image. It uses the Toolkit class to load the rat.gif from the default working directory. It implements various getter and setter methods to manipulate its coordinates and retrieve its image.

The Floor class extends the Canvas class as called for in the design. It contains a Vector reference field that is initialized to the Vector reference passed in via the constructor. The Floor class overrides the paint() method to draw the rat images contained in the Vector object upon itself.

The RobotRat class has a lot going on. It uses a static initializer to initialize the JFrame, Vector, and Floor objects. The JFrame size and location is set in the static initializer as well. Because these statements are located in a static initializer block they are executed when the class is loaded. The RobotRat constructor creates an instance of a Rat and adds it to the vector of rats via the _rats reference. The repaint() method is then called via the _floor reference. This causes the paint() method to be called which draws the collection of Rat images on the floor.

The NetRatServer class is used to test the first iteration code. For testing purposes a simple GUI that consists of one button is created that allows users to move a couple of rats about the floor in limited directions.

TESTING THE FIRST ITERATION CODE

Figures 20-11 and 20-12 show the NetRatServer application on start-up and after the move button has been clicked about ten times.

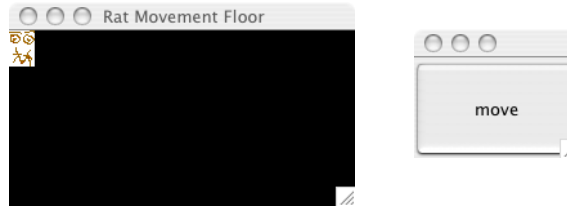


Figure 20-11: NetRatServer Application Upon Start-up



Figure 20-12: NetRatServer Application After Approximately 10 move Button Clicks

Quick Review

When ready to proceed with the implementation phase select the smallest subset of the design that results in a set of application functionality that can be tested. Some code written during the implementation may be ultimately discarded before the final application is released.

IMPLEMENTING NETRATSERVER - SECOND ITERATION

The second development iteration will focus on implementing RMI functionality. Table 20-5 gives the design considerations and decisions for this iteration.

Check-Off	Design Considerations	Design Decisions
	Objectives for the second iteration	Implement Remote Method Invocation (RMI) functionality. This will entail creating an interface called RobotRatInterface that contains all the method declarations currently implemented in the RobotRat class. Once the RobotRatInterface is complete the RobotRat class needs to be retrofitted to implement RobotRatInterface and extend UnicastRemoteObject. The rmic tool then needs to be run to create the required RobotRat stub class. A slight modification will also have to be made to the NetRatServer class in order to utilize the RMI version of RobotRat.
	RobotRatInterface	The RobotRatInterface must extend the java.rmi.Remote interface and provide method declarations for all the move methods currently defined in the RobotRat class.
	RobotRat class	The RobotRat class must extend the java.rmi.server.UnicastRemoteObject and implement the RobotRatInterface.
	rmic tool	The rmic tool needs to be run against the RobotRat class file to create the RobotRat_stub.class file.

Table 20-5: Second Iteration Design Considerations and Decisions

Check-Off	Design Considerations	Design Decisions
	NetRatServer	The NetRatServer needs to be slightly modified to properly handle the RMI version of the RobotRat class. This will entail wrapping RobotRat method calls in try-catch blocks and adding the code to start the registry and bind an instance of RobotRat to a service name.

Table 20-5: Second Iteration Design Considerations and Decisions

SECOND ITERATION CODE

The following code is written or modified during the second iteration:

```

1      import java.rmi.*;
2
3      interface RobotRatInterface extends Remote {
4
5          public void moveEast() throws RemoteException;
6          public void moveSouth() throws RemoteException;
7          public void moveWest() throws RemoteException;
8          public void moveNorth() throws RemoteException;
9          public void moveNorthWest() throws RemoteException;
10         public void moveSouthWest() throws RemoteException;
11         public void moveNorthEast() throws RemoteException;
12         public void moveSouthEast() throws RemoteException;
13
14     }

```

20.7 RobotRatInterface.java

```

1      import javax.swing.*;
2      import java.util.*;
3      import java.rmi.*;
4      import java.rmi.server.*;
5
6      public class RobotRat extends UnicastRemoteObject implements RobotRatInterface {
7
8          private static JFrame _frame = null;
9          private static Floor _floor = null;
10         private static Vector _rats = null;
11         private Rat _its_rat = null;
12
13         static {
14             _rats = new Vector();
15             _floor = new Floor(_rats);
16             _frame = new JFrame("Rat Movement Floor");
17             _frame.getContentPane().add(_floor);
18             _frame.setSize(300, 300);
19             _frame.setLocation(200, 200);
20             _frame.show();
21         }
22
23
24         public RobotRat() throws RemoteException {
25             _its_rat = new Rat();
26             _rats.addElement(_its_rat);
27             _floor.repaint();
28         }
29
30         public void moveEast(){
31             _its_rat.setX(_its_rat.getX() + 3);
32             _floor.repaint();
33         }
34
35         public void moveSouth(){
36             _its_rat.setY(_its_rat.getY() + 3);
37             _floor.repaint();
38         }
39
40         public void moveWest(){
41             _its_rat.setX(_its_rat.getX() - 3);
42             _floor.repaint();
43         }
44
45         public void moveNorth(){
46             _its_rat.setY(_its_rat.getY() - 3);

```

20.8 RobotRat.java (mod 1)

```

47     _floor.repaint();
48 }
49
50 public void moveNorthWest() {
51     _its_rat.setY(_its_rat.getY() - 3);
52     _its_rat.setX(_its_rat.getX() - 3);
53     _floor.repaint();
54 }
55
56 public void moveSouthWest() {
57     _its_rat.setY(_its_rat.getY() + 3);
58     _its_rat.setX(_its_rat.getX() - 3);
59     _floor.repaint();
60 }
61
62 public void moveNorthEast() {
63     _its_rat.setY(_its_rat.getY() - 3);
64     _its_rat.setX(_its_rat.getX() + 3);
65     _floor.repaint();
66 }
67
68 public void moveSouthEast() {
69     _its_rat.setY(_its_rat.getY() + 3);
70     _its_rat.setX(_its_rat.getX() + 3);
71     _floor.repaint();
72 }
73 } // end RobotRat class definition

```

20.9 NetRatServer.java (mod 1)

```

1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.rmi.*;
5  import java.rmi.registry.*;
6  import java.net.*;
7
8  public class NetRatServer extends JFrame implements ActionListener {
9
10     private JButton button1 = null;
11     private RobotRatInterface _r1 = null;
12     private RobotRatInterface _r2 = null;
13
14     public NetRatServer() {
15         try {
16             _r1 = new RobotRat();
17             _r2 = new RobotRat();
18         } catch (Exception ignored) { ignored.printStackTrace(); }
19         button1 = new JButton("move");
20         button1.addActionListener(this);
21         this.getContentPane().add(button1);
22         this.setSize(100, 100);
23         this.setLocation(300, 300);
24         this.show();
25     }
26
27     public void actionPerformed(ActionEvent ae) {
28         if (ae.getActionCommand().equals("move")) {
29             try {
30                 _r1.moveEast();
31                 _r2.moveSouthEast();
32             } catch (Exception ignored) { }
33         }
34     }
35
36     public static void main(String[] args) {
37         System.out.println("NetRatServer Lives!!");
38         new NetRatServer();
39         try {
40             System.out.println("Starting registry...");
41             LocateRegistry.createRegistry(1099);
42             System.out.println("Registry started on port 1099.");
43             System.out.println("Binding service name to remote object...");
44             Naming.bind("Robot_Rat", new RobotRat());
45             System.out.println("Bind successful.");
46             System.out.println("Ready for remote method invocation.");
47         } catch (Exception e) {
48             e.printStackTrace();
49         }
50     }
51 } // end NetRatServer class definition

```

Referring to example 20.7 through 20.9 — the `RobotRatInterface` extends the `java.rmi.Remote` interface and provides a declaration for each of the move methods currently defined in the `RobotRat` class.

The `RobotRat` class extends the `java.rmi.server.UnicastRemoteObject` and implements `RobotRatInterface`. Notice that the only modifications required to `RobotRat` include the extended class declaration and the addition of a throws clause to the constructor.

The `NetRatServer` class has been modified in several ways to accommodate the RMI-enabled `RobotRat` class. First, the `RobotRatInterface` is used on lines 11 and 12 to declare references `r1` and `r2`. Next, an instance of a registry is started programmatically in the `main()` method on line 41, and on line 44 an instance of `RobotRat` is bound to the service name “`Robot_Rat`”.

TESTING SECOND ITERATION CODE

Before you can test this version of `NetRatServer` the `rmic` tool must be run against the compiled `RobotRat` class file to generate the `RobotRat_stub.class` file. This file must then be deployed with the rest of the server code. Figure 20-13 shows the `NetRatServer` terminal window output and floor at application start-up. Figure 20-14 shows how the floor looks after approximately ten clicks of the move button.

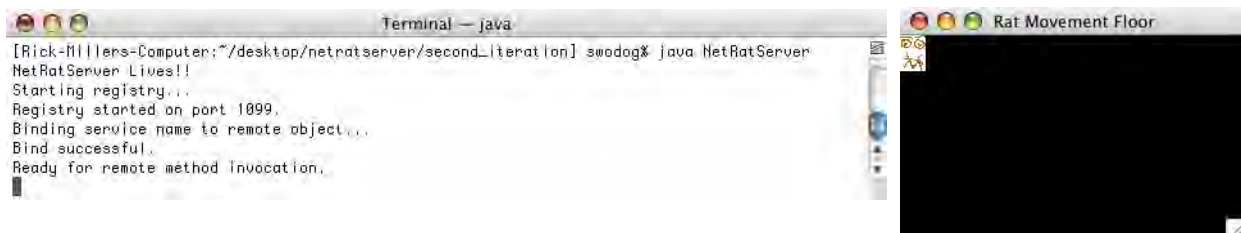


Figure 20-13: RMI-Enabled NetRatServer Application at Start-up



Figure 20-14: RMI-Enabled NetRatServer Application After Approximately 10 move Button Clicks

Referring to figure 20-14 — notice now there are three rat images displayed on the floor: the two that move to the right when the move button is clicked, and the third in the upper left corner that corresponds to the instance of `RobotRat` that was bound to the service name “`Robot_Rat`” in the body of the `main()` method.

To fully test the `NetRatServer` RMI functionality it is now necessary to create the RMI-based client application.

Quick Review

Remote Method Invocation functionality is often easier to implement than socket-based functionality. This is because the RMI runtime handles socket communication and thread management issues behind the scenes. Implementing RMI functionality, however, requires the programmer to generate the necessary stub files for deployment with the server and client applications. RMI servers must bind an instance of the RMI object to a service name in an RMI registry.

The RMI-Based Client

This section presents an RMI-based client application that can send commands to a RobotRat object via a reference retrieved from the server's RMI registry.

20.10 RMI_NetRatClient.java

```

1      import java.rmi.*;
2      import java.awt.*;
3      import java.awt.event.*;
4      import javax.swing.*;
5
6      public class RMI_NetRatClient extends JFrame implements ActionListener {
7
8          private JButton _button1 = null;
9          private JButton _button2 = null;
10         private JButton _button3 = null;
11         private JButton _button4 = null;
12         private JButton _button5 = null;
13         private JButton _button6 = null;
14         private JButton _button7 = null;
15         private JButton _button8 = null;
16         private JButton _button9 = null;
17         private RobotRatInterface _robot_rat = null;
18
19
20         public RMI_NetRatClient(String host){
21             super("Robot Rat Control Panel");
22
23             try{
24                 _robot_rat = (RobotRatInterface)Naming.lookup("rmi://" + host + "/Robot_Rat");
25
26                 }catch(Exception e){
27                     e.printStackTrace();
28                 }
29             this.setUpGui();
30         }
31
32         public void actionPerformed(ActionEvent ae){
33             try{
34                 if(ae.getActionCommand().equals("N")){
35                     _robot_rat.moveNorth();
36                 }else if(ae.getActionCommand().equals("NE")){
37                     _robot_rat.moveNorthEast();
38                 }else if(ae.getActionCommand().equals("E")){
39                     _robot_rat.moveEast();
40                 }else if(ae.getActionCommand().equals("SE")){
41                     _robot_rat.moveSouthEast();
42                 }else if(ae.getActionCommand().equals("S")){
43                     _robot_rat.moveSouth();
44                 }else if(ae.getActionCommand().equals("SW")){
45                     _robot_rat.moveSouthWest();
46                 }else if(ae.getActionCommand().equals("W")){
47                     _robot_rat.moveWest();
48                 }else if(ae.getActionCommand().equals("NW")){
49                     _robot_rat.moveNorthWest();
50                 }
51
52             }catch(RemoteException re){
53                 System.out.println("actionPerformed(): problem calling remote robot_rat method.");
54                 re.printStackTrace();
55             }
56         } // end constructor
57
58         public void setUpGui(){
59             _button1 = new JButton("NW");
60             _button1.addActionListener(this);
61
62             _button2 = new JButton("N");
63             _button2.addActionListener(this);
64
65             _button3 = new JButton("NE");
66             _button3.addActionListener(this);
67
68             _button4 = new JButton("W");
69             _button4.addActionListener(this);
70
71             _button5 = new JButton("");

```

```

72     _button5.addActionListener(this);
73
74     _button6 = new JButton("E");
75     _button6.addActionListener(this);
76
77     _button7 = new JButton("SW");
78     _button7.addActionListener(this);
79
80     _button8 = new JButton("S");
81     _button8.addActionListener(this);
82
83     _button9 = new JButton("SE");
84     _button9.addActionListener(this);
85
86     this.getContentPane().setLayout(new GridLayout(3,3,0,0));
87     this.getContentPane().add(_button1);
88     this.getContentPane().add(_button2);
89     this.getContentPane().add(_button3);
90     this.getContentPane().add(_button4);
91     this.getContentPane().add(_button5);
92     this.getContentPane().add(_button6);
93     this.getContentPane().add(_button7);
94     this.getContentPane().add(_button8);
95     this.getContentPane().add(_button9);
96     this.setSize(200, 200);
97     this.setLocation(300, 300);
98     this.pack();
99     this.show();
100 } // end setUpGui()
101
102
103 public static void main(String[] args){
104     try{
105         new RMI_NetRatClient(args[0]);
106     }catch(ArrayIndexOutOfBoundsException e1){
107         System.out.println("Usage: java NetRatClient <host>");
108     }
109     catch(Exception e2){
110         e2.printStackTrace();
111     }
112 } // end main()
113 } // end RMI_NetRatClient class definition

```

Referring to example 20-10 — the `RMI_NetRatClient` class allows the user to send commands to a remote `RobotRat`. Users enter movement commands via a grid of `JButtons`. The constructor method looks up the `Robot_Rat` service on the server at the designated IP address.

To test the `RMI_NetRatClient` code the server must be running on the local machine or remote server. Figure 20-15 shows the `RMI_NetRatClient` application running.

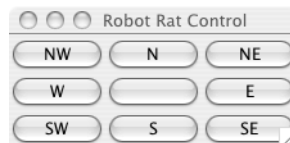


Figure 20-15: `RMI_NetRatClient` Application

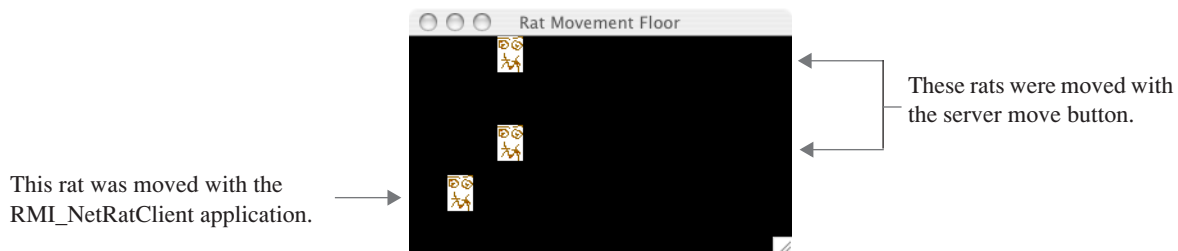


Figure 20-16: The Floor After Testing `RMI_NetRatClient`.

TESTING MULTIPLE RMI CLIENTS

The RMI_NetRatClient application can be used to test multiple RMI client connections. To do so simply start up another instance of the RMI_NetRatClient application. Given the current NetRatServer design, testing reveals that multiple RMI clients control only one RobotRat around the floor, which is the one bound to the service name Robot_Rat in the body of the NetRatServer main() method. What's needed is a modification to the server application design that will allow each RMI client to control their very own RobotRat. This is the objective of the third NetRat-Server development iteration.

QUICK REVIEW

RMI-based clients gain access to an RMI object via its service name. The remote object's interface and stub classes must be deployed with the client application. Calls to the remote object look like ordinary method calls. All network communication required between the RMI-based client and remote object is handled by the RMI runtime environment.

NETRATSERVER IMPLEMENTATION – THIRD ITERATION

The current design of the robot rat server application results in all RMI clients controlling the same instance of a RobotRat. What's needed is some way to create a new instance of a RobotRat for each RMI client connection. This can easily be done by creating another class on the server side whose job is to create RobotRat objects. We call such a class a *factory class*. Since our factory class will create RobotRat objects a good name for the class is RobotRatFactory. Table 20-6 presents the design considerations and design decisions that will guide this development iteration.

Check-Off	Design Consideration	Design Decision
	Objectives for third iteration server application development activities.	Modify the server design to incorporate a RobotRatFactory class so that RMI clients can manipulate their own RobotRat.
	RobotRatFactory class	Design a new class named RobotRatFactory. This class will be an RMI class like RobotRat which means a RobotRatFactory interface must also be created. Clients will get an instance of RobotRatFactory and use it to retrieve a RobotRat object from the server.
	RobotRatFactory interface	The RobotRatFactory interface needs to be created to support the RobotRatFactory class. The name of this interface will be RobotRatFactoryInterface. RobotRatFactoryInterface will extend the java.rmi.remote interface.
	NetRatServer class	The NetRatServer class will need to be modified to bind an instance of RobotRatFactory to a service name. The name of the service can be Robot_Rat_Factory.
	RMI_NetRatClient class	The RMI_Client will need to be modified to use the RobotRatFactory.
	rmic tool	The RobotRatFactory_stub.class file will need to be generated with the rmic tool and deployed to the server side and client side.

Table 20-6: Third Iteration Design Considerations and Decisions

UPDATED SERVER APPLICATION CLASS DIAGRAM

Figure 20-17 offers the modified version of the robot rat server application class diagram. The NetRatServer class will now have a dependency on the RobotRatFactory class. There is still a dependency between NetRatServer

and RobotRat but only because of the earlier test code. There is also a dependency between NetRatServer and RobotRatInterface because of the same test code. All test code can be removed in a later iteration to eliminate unnecessary (*residual*) dependencies.

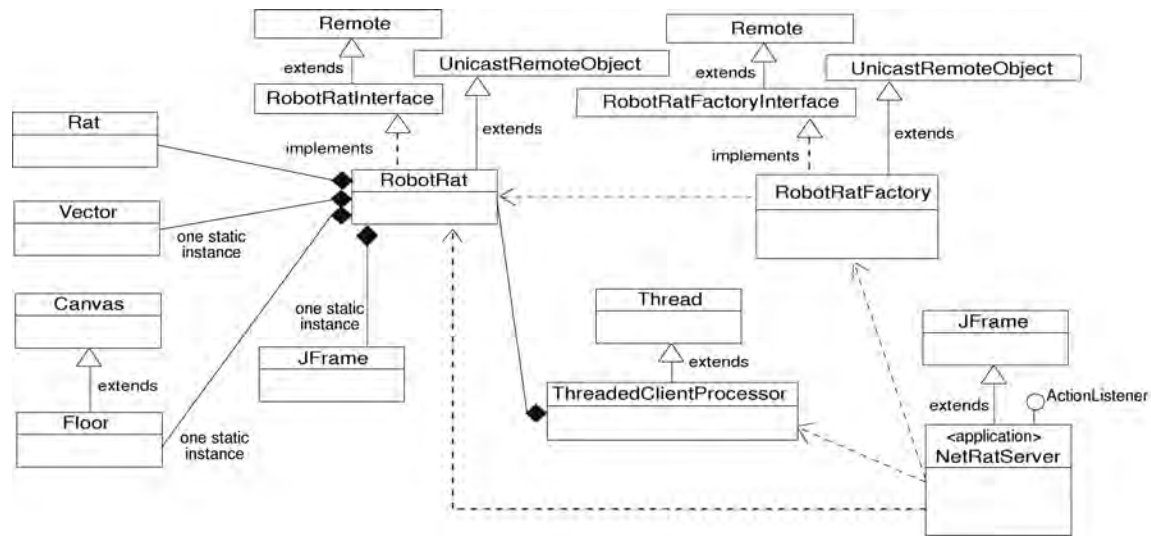


Figure 20-17: Updated Robot Rat Server Application Class Diagram

THIRD ITERATION CODE

The following new classes and modified code are the result of third iteration development activities.

```

1      import java.rmi.*;
2
3      interface RobotRatFactoryInterface extends Remote {
4          public RobotRatInterface getRobotRat() throws RemoteException;
5      }

```

20.11 RobotRatFactoryInterface.java

```

1      import java.rmi.*;
2      import java.rmi.server.*;
3
4      public class RobotRatFactory extends UnicastRemoteObject
5          implements RobotRatFactoryInterface {
6
7          public RobotRatFactory() throws RemoteException {
8              System.out.println("RobotRatFactoryImplementation object created!");
9          }
10
11         public RobotRatInterface getRobotRat() throws RemoteException {
12             RobotRatInterface new_rat = null;
13             try{
14                 new_rat = new RobotRat();
15             }catch(RemoteException re){
16                 System.out.println("getRobotRat(): problem creating new RobotRat object!");
17                 re.printStackTrace();
18             }
19             return new_rat;
20         }
21     } // end RobotRatFactory class definition

```

20.12 RobotRatFactory.java

```

1      import javax.swing.*;
2      import java.awt.*;
3      import java.awt.event.*;
4      import java.rmi.*;
5      import java.rmi.registry.*;
6      import java.net.*;
7
8      public class NetRatServer extends JFrame implements ActionListener {
9

```

20.13 NetRatServer.java (mod 2)

```

10     private JButton button1 = null;
11     private RobotRatInterface _r1 = null;
12     private RobotRatInterface _r2 = null;
13
14     public NetRatServer(){
15         try{
16             _r1 = new RobotRat();
17             _r2 = new RobotRat();
18         }catch(Exception ignored){ ignored.printStackTrace(); }
19         button1 = new JButton("move");
20         button1.addActionListener(this);
21         this.getContentPane().add(button1);
22         this.setSize(100, 100);
23         this.setLocation(300, 300);
24         this.show();
25     }
26
27     public void actionPerformed(ActionEvent ae){
28         if(ae.getActionCommand().equals("move")){
29             try{
30                 _r1.moveEast();
31                 _r2.moveSouthEast();
32             }catch(Exception ignored){ }
33         }
34     }
35
36     public static void main(String[] args){
37         System.out.println("NetRatServer Lives!!");
38         new NetRatServer();
39         try{
40             System.out.println("Starting registry...");
41             LocateRegistry.createRegistry(1099);
42             System.out.println("Registry started on port 1099.");
43             System.out.println("Binding service name to remote object...");
44             Naming.bind("Robot_Rat_Factory", new RobotRatFactory());
45             System.out.println("Bind successful.");
46             System.out.println("Ready for remote method invocation.");
47         }catch(Exception e){
48             e.printStackTrace();
49         }
50     } // end main() method
51 } // end NetRatServer class

```

Referring to example 20.11 through 20.13 — `RobotRatFactoryInterface` declares one public method named `getRobotRat()` which returns a reference to an object of type `RobotRatInterface`. The `RobotRatFactory` class implements `RobotRatFactoryInterface` and provides functionality for the `getRobotRat()` method. All the `getRobotRat()` method does is create and return a new instance of `RobotRat`. This has significant implications for the behavior of the server application — all RMI clients will have their very own rat to move around which is what we desire.

The `NetRatServer` class is modified on line 44 to bind an instance of `RobotRatFactory` with the `Robot_Rat_Factory` service name.

After these classes are compiled the `rmic` tool must be run to generate the `RobotRatFactory_stub.class` file. The `RMI_NetRatClient` application will now need four server-side class files to operate properly. These include `RobotRatInterface.class`, `RobotRat_stub.class`, `RobotRatFactoryInterface.class`, and `RobotRatFactory_stub.class`.

TESTING THE THIRD ITERATION CODE

Compile the code and generate the necessary stub files with the `rmic` tool. You can now crank up the robot rat server application to see if everything works. However, to fully test the server application you must make a modification to the `RMI_NetRatClient` application so that it can utilize a `RobotRatFactory` to get an instance of a `RobotRat`. The modified client application is given in example 20.14.

20.14 RMI_NetRatClient.java (mod 1)

```

1     import java.rmi.*;
2     import java.awt.*;
3     import java.awt.event.*;
4     import javax.swing.*;
5
6     public class RMI_NetRatClient extends JFrame implements ActionListener {
7
8         private JButton _button1 = null;
9         private JButton _button2 = null;
10        private JButton _button3 = null;

```

```

11     private JButton _button4 = null;
12     private JButton _button5 = null;
13     private JButton _button6 = null;
14     private JButton _button7 = null;
15     private JButton _button8 = null;
16     private JButton _button9 = null;
17     private RobotRatInterface _robot_rat = null;
18     private RobotRatFactoryInterface _robot_rat_factory = null;
19
20
21     public RMI_NetRatClient(String host){
22         super("Robot Rat Control Panel");
23
24         try{
25             _robot_rat_factory = (RobotRatFactoryInterface)Naming.lookup("rmi://" + host +
26                                                                 "/Robot_Rat_Factory");
27             _robot_rat = _robot_rat_factory.getRobotRat();
28
29             }catch(Exception e){
30                 e.printStackTrace();
31             }
32         this.setUpGui();
33     }
34
35     public void actionPerformed(ActionEvent ae){
36         try{
37             if(ae.getActionCommand().equals("N")){
38                 _robot_rat.moveNorth();
39             }else if(ae.getActionCommand().equals("NE")){
40                 _robot_rat.moveNorthEast();
41             }else if(ae.getActionCommand().equals("E")){
42                 _robot_rat.moveEast();
43             }else if(ae.getActionCommand().equals("SE")){
44                 _robot_rat.moveSouthEast();
45             }else if(ae.getActionCommand().equals("S")){
46                 _robot_rat.moveSouth();
47             }else if(ae.getActionCommand().equals("SW")){
48                 _robot_rat.moveSouthWest();
49             }else if(ae.getActionCommand().equals("W")){
50                 _robot_rat.moveWest();
51             }else if(ae.getActionCommand().equals("NW")){
52                 _robot_rat.moveNorthWest();
53             }
54
55             }catch(RemoteException re){
56                 System.out.println("actionPerformed(): problem calling remote robot_rat method.");
57                 re.printStackTrace();
58             }
59         }
60
61     public void setUpGui(){
62         _button1 = new JButton("NW");
63         _button1.addActionListener(this);
64
65         _button2 = new JButton("N");
66         _button2.addActionListener(this);
67
68         _button3 = new JButton("NE");
69         _button3.addActionListener(this);
70
71         _button4 = new JButton("W");
72         _button4.addActionListener(this);
73
74         _button5 = new JButton("");
75         _button5.addActionListener(this);
76
77         _button6 = new JButton("E");
78         _button6.addActionListener(this);
79
80         _button7 = new JButton("SW");
81         _button7.addActionListener(this);
82
83         _button8 = new JButton("S");
84         _button8.addActionListener(this);
85
86         _button9 = new JButton("SE");
87         _button9.addActionListener(this);
88
89         this.getContentPane().setLayout(new GridLayout(3,3,0,0));
90         this.getContentPane().add(_button1);
91

```

```

92     this.getContentPane().add(_button2);
93     this.getContentPane().add(_button3);
94     this.getContentPane().add(_button4);
95     this.getContentPane().add(_button5);
96     this.getContentPane().add(_button6);
97     this.getContentPane().add(_button7);
98     this.getContentPane().add(_button8);
99     this.getContentPane().add(_button9);
100    this.setSize(200, 200);
101    this.setLocation(300, 300);
102    this.pack();
103    this.show();
104    } // end setUpGui() method
105
106    public static void main(String[] args){
107        try{
108            new RMI_NetRatClient(args[0]);
109        }catch(ArrayIndexOutOfBoundsException e1){
110            System.out.println("Usage: java NetRatClient <host>");
111        }
112        catch(Exception e2){
113            e2.printStackTrace();
114        }
115    } // end main()
116
117    } // end NetRatClient class definition

```

Referring to example 20.14 — the `RMI_NetRatClient` application must now retrieve a `RobotRatFactory` object from the server and use it to get a reference to a `RobotRat` object. `RobotRatInterface` and `RobotRatFactoryInterface` are used on lines 17 and 18 to declare the references `_robot_rat` and `_robot_rat_factory` respectively. On lines 25 and 26 a `RobotRatFactory` reference is retrieved from the server and assigned to the reference variable `_robot_rat_factory`. On line 27 the `getRobotRat()` method is called via the `_robot_rat_factory` reference to retrieve a `RobotRat` reference. Method calls to the server-side `RobotRat` object are made via the `_robot_rat` reference variable as before.

TESTING MULTIPLE RMI CLIENT APPLICATIONS

Now that the necessary modifications have been made to the server and client applications multiple RMI client connections can be tested. Start-up the `NetRatServer` application and use its move button to move the two server-controlled robot rats out of the way. Figure 20-18 shows the floor after the server-controlled robot rats have been moved out of the way.

Notice there are only two rats visible when the server starts up.



Figure 20-18: The Floor After Approximately 15 Clicks of the Server-Side move Button

Referring to figure 20-18 — notice now that only the two server-side test robot rats are drawn on the floor when the `NetRatServer` application starts up.

With the server running, launch the modified version of the `RMI_NetRatClient` application. Figure 20-19 shows the server floor after one client has been started and has moved its robot rat several clicks south. Now start-up another instance of the `RMI_NetRatClient` application. Figure 20-20 shows the server floor now populated with two RMI-client-controlled and two server-controlled robot rats. To fully test the application(s) up to this point deploy the server and client on separate computers and test remotely. It's time now to implement the last bit of client-server functionality.

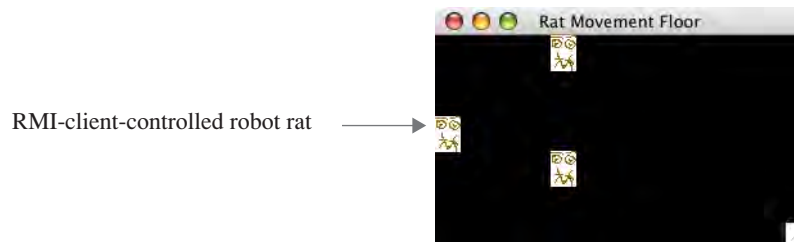


Figure 20-19: Server Floor After RMI-Client-Controlled Robot Rat Moves South Several Clicks

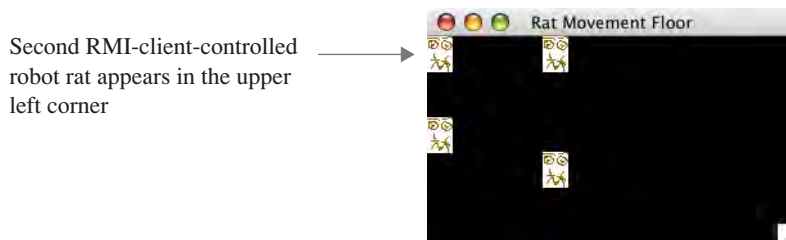


Figure 20-20: Server Floor After Second RMI-Client-Controlled Robot Rat Appears

Quick Review

The RMI runtime manages client access to RMI server-side objects. In cases where RMI-based clients must have access to dedicated server-side remote objects a factory class can be employed to create the dedicated remote objects as required. RMI-based clients must first get a reference to the server-side factory object and use it to create the dedicated remote object it requires. This solves the dedicated remote object problem but raises the level of application complexity. Now, in addition to the dedicated remote object interface and stub classes, the factory interface and stub classes must also be deployed with the server and client applications.

NETRATSERVER IMPLEMENTATION – FOURTH ITERATION

There is only one piece of the robot rat server application design left to implement and that is the ability for it to handle multiple socket-based client connections. To do this it must be a multi-threaded server. Also, since socket-based communication takes place at a lower level than RMI method calls, a client-server protocol must be established to facilitate client-server communication. Table 20-7 presents the design considerations and decisions that apply to the robot rat server application fourth iteration development activities.

Check-Off	Design Consideration	Design Decision
	Objectives for fourth iteration server application development activities.	The objective of the fourth iteration development activities is to modify the robot rat server application so it can process multiple socket-based client connections. To do this the <code>ThreadedClientProcessor</code> class must be created. The <code>NetRatServer</code> class must also be modified to listen for incoming socket-based client connections and pass these connections to an instance of <code>ThreadedClientProcessor</code> for further processing.

Table 20-7: Third Iteration Design Considerations and Decisions

Check-Off	Design Consideration	Design Decision
	ThreadedClientProcessor class	The ThreadedClientProcessor class will extend the Thread Class. Its responsibility will be to process client communication for as long as the client application is connected to the server. It will take an instance of java.net.Socket as an argument to its constructor. The Socket object will be used to obtain the InputStream and OutputStream objects through which client-server communication will occur. The ThreadedClientProcessor class is also responsible for translating client data into robot rat movement commands.
	NetRatServer class	The NetRatServer class must be modified to listen for incoming socket-based client connections. When an incoming client connections is detected the resulting Socket object will be passed off to an instance of ThreadedClientProcessor for continued processing.
	client-server protocol	A simple protocol must be established that allows data to be sent from the client application and translated into robot rat commands. The requirements of the robot rat application are simple. The client will send the integer values 0 through 7 to the server in response to user button clicks. The integer values will be mapped in the ThreadedClientProcessor to RobotRat movement method calls according the following mapping: <pre style="margin-left: 40px;"> NORTH = 0; NORTH_EAST = 1; EAST = 2; SOUTH_EAST = 3; SOUTH = 4; SOUTH_WEST = 5; WEST = 6; NORTH_WEST = 7; </pre>

Table 20-7: Third Iteration Design Considerations and Decisions

FOURTH ITERATION CODE

The following new and modified code is the result of fourth iteration development activities.

20.15 ThreadedClientProcessor.java

```

1      import java.io.*;
2      import java.net.*;
3
4      public class ThreadedClientProcessor extends Thread {
5
6          private Socket          _socket      = null;
7          private DataInputStream _dis        = null;
8          private DataOutputStream _dos       = null;
9          private RobotRatInterface _robot_rat = null;
10
11         private static final int NORTH      = 0;
12         private static final int NORTH_EAST = 1;
13         private static final int EAST       = 2;
14         private static final int SOUTH_EAST = 3;
15         private static final int SOUTH     = 4;
16         private static final int SOUTH_WEST = 5;
17         private static final int WEST      = 6;
18         private static final int NORTH_WEST = 7;
19
20
21         public ThreadedClientProcessor(Socket socket){
22             _socket = socket;
23             try{
24                 _dos = new DataOutputStream(_socket.getOutputStream());
25                 _dis = new DataInputStream(_socket.getInputStream());
26                 _robot_rat = new RobotRat();
27             }catch(Exception e){

```

```

28         System.out.println("ThreadedClientProcessor: Problem creating IOStream objects!");
29
30         e.printStackTrace();
31     }
32 } // end constructor
33
34
35 public void run(){
36     int command = 0;
37     try{
38         while((command = _dis.readInt()) != -1){
39             switch(command){
40                 case NORTH      : _robot_rat.moveNorth();
41                                 _dos.writeUTF("Rat moved North");
42                                 break;
43                 case NORTH_EAST : _robot_rat.moveNorthEast();
44                                 _dos.writeUTF("Rat moved North-East");
45                                 break;
46                 case EAST       : _robot_rat.moveEast();
47                                 _dos.writeUTF("Rat moved East");
48                                 break;
49                 case SOUTH_EAST : _robot_rat.moveSouthEast();
50                                 _dos.writeUTF("Rat moved South-East");
51                                 break;
52                 case SOUTH      : _robot_rat.moveSouth();
53                                 _dos.writeUTF("Rat moved South");
54                                 break;
55                 case SOUTH_WEST : _robot_rat.moveSouthWest();
56                                 _dos.writeUTF("Rat moved South-West");
57                                 break;
58                 case WEST       : _robot_rat.moveWest();
59                                 _dos.writeUTF("Rat moved West");
60                                 break;
61                 case NORTH_WEST : _robot_rat.moveNorthWest();
62                                 _dos.writeUTF("Rat moved North-West");
63                                 break;
64                 default         : _dos.writeUTF("Invalid command");
65
66
67             } // end switch
68         } // end while
69     } catch (EOFException ignored) { }
70     catch (Exception e) {
71         e.printStackTrace();
72     }
73     finally{
74         try{
75             _socket.close();
76         } catch (Exception ignored) { }
77     }
78
79     try{
80         _socket.close();
81     } catch (Exception ignored) { }
82 } // end run()
83 } // end ThreadedClientProcessor class definition

```

20.16 NetRatServer.java (mod 3)

```

1     import javax.swing.*;
2     import java.awt.*;
3     import java.awt.event.*;
4     import java.rmi.*;
5     import java.rmi.registry.*;
6     import java.net.*;
7
8     public class NetRatServer extends JFrame implements ActionListener {
9
10        private JButton button1 = null;
11        private RobotRatInterface _r1 = null;
12        private RobotRatInterface _r2 = null;
13
14        public NetRatServer(){
15            try{
16                _r1 = new RobotRat();
17                _r2 = new RobotRat();
18            } catch (Exception ignored) { ignored.printStackTrace(); }
19            button1 = new JButton("move");
20            button1.addActionListener(this);
21            this.getContentPane().add(button1);
22            this.setSize(100, 100);

```



```

23         this.setLocation(300, 300);
24         this.show();
25     }
26
27     public void actionPerformed(ActionEvent ae){
28         if(ae.getActionCommand().equals("move")){
29             try{
30                 _r1.moveEast();
31                 _r2.moveSouthEast();
32             }catch(Exception ignored){ }
33         }
34     }
35
36     public static void main(String[] args){
37         System.out.println("NetRatServer Lives!!");
38         new NetRatServer();
39         try{
40             System.out.println("Starting registry...");
41             LocateRegistry.createRegistry(1099);
42             System.out.println("Registry started on port 1099.");
43             System.out.println("Binding service name to remote object...");
44             Naming.bind("Robot_Rat_Factory", new RobotRatFactory());
45             System.out.println("Bind successful.");
46             System.out.println("Ready for remote method invocation.");
47
48             System.out.println("Creating ServerSocket Object...");
49             ServerSocket server_socket = new ServerSocket(5001);
50             System.out.println("ServerSocket object created successfully!");
51             while(true){
52                 System.out.println("Listening for incoming client connections...");
53                 Socket socket = server_socket.accept();
54                 System.out.println("Incoming client connection detected.");
55                 System.out.println("Creating ThreadedClientProcessor object...");
56                 ThreadedClientProcessor client_processor = new ThreadedClientProcessor(socket);
57                 System.out.println("ThreadedClientProcessor object created.");
58                 System.out.println("Calling start() method...");
59                 client_processor.start();
60             }
61         }catch(Exception e){
62             e.printStackTrace();
63         }
64     }
65 } // end NetRatServer class

```

Referring to examples 20.15 and 20.16 — the `ThreadedClientProcessor` class takes a `Socket` reference as an argument to its constructor. The `Socket` reference is used to obtain the `InputStream` and `OutputStream` references. The `OutputStream` reference is used on line 24 to create the `DataOutputStream` object and the `InputStream` reference is used on line 25 to create the `DataInputStream` object. If lines 24 and 25 execute normally then line 26 will execute creating the instance of `RobotRat` that will be moved about the screen during this client session.

The bulk of the processing in the `ThreadedClientProcessor` class takes place in the body of the `run()` method. Client commands are read with the `DataInputStream` reference `_dis`. Client commands are translated into `RobotRat` move commands in the body of the switch statement according to the established protocol. In response to each client command the `ThreadedClientProcessor` will respond to the client with a UTF string indicating which way the `RobotRat` moved.

Turn your attention now to the modified `NetRatServer` class. Additional code was added starting on line 48 that enables the `NetRatServer` application to listen for incoming client connections using a `ServerSocket` object. In this example the server will listen for incoming client connections on port 5001. The while statement starting on line 51 performs the bulk of the processing, continuously listening for incoming client connections and passing the resulting `Socket` reference to the constructor of a `ThreadedClientProcessor` object and then calling its `start()` method.

TESTING FOURTH ITERATION CODE

To fully test the latest version of the `NetRatServer` application you'll need to code up a socket-based client application. Example 20.17 gives the code for the `Socket_NetRatClient` class that allows you to connect to the robot rat server application using sockets.

```

1      import java.net.*;
2      import java.io.*;
3      import java.awt.*;
4      import java.awt.event.*;
5      import javax.swing.*;
6
7      public class Socket_NetRatClient extends JFrame implements ActionListener {
8
9          private JButton _button1 = null;
10         private JButton _button2 = null;
11         private JButton _button3 = null;
12         private JButton _button4 = null;
13         private JButton _button5 = null;
14         private JButton _button6 = null;
15         private JButton _button7 = null;
16         private JButton _button8 = null;
17         private JButton _button9 = null;
18         private Socket _socket = null;
19         private DataInputStream _dis = null;
20         private DataOutputStream _dos = null;
21
22         private static final int NORTH = 0;
23         private static final int NORTH_EAST = 1;
24         private static final int EAST = 2;
25         private static final int SOUTH_EAST = 3;
26         private static final int SOUTH = 4;
27         private static final int SOUTH_WEST = 5;
28         private static final int WEST = 6;
29         private static final int NORTH_WEST = 7;
30
31
32         public Socket_NetRatClient(String host){
33             super("Robot Rat Control Panel");
34
35             try{
36                 _socket = new Socket(host, 5001);
37                 _dos = new DataOutputStream(_socket.getOutputStream());
38                 _dis = new DataInputStream(_socket.getInputStream());
39             }catch(Exception e){
40                 e.printStackTrace();
41             }
42             this.setUpGui();
43         }
44
45         public void actionPerformed(ActionEvent ae){
46             try{
47                 if(ae.getActionCommand().equals("N")){
48                     _dos.writeInt(NORTH);
49                     System.out.println(_dis.readUTF());
50                 }else if(ae.getActionCommand().equals("NE")){
51                     _dos.writeInt(NORTH_EAST);
52                     System.out.println(_dis.readUTF());
53                 }else if(ae.getActionCommand().equals("E")){
54                     _dos.writeInt(EAST);
55                     System.out.println(_dis.readUTF());
56                 }else if(ae.getActionCommand().equals("SE")){
57                     _dos.writeInt(SOUTH_EAST);
58                     System.out.println(_dis.readUTF());
59                 }else if(ae.getActionCommand().equals("S")){
60                     _dos.writeInt(SOUTH);
61                     System.out.println(_dis.readUTF());
62                 }else if(ae.getActionCommand().equals("SW")){
63                     _dos.writeInt(SOUTH_WEST);
64                     System.out.println(_dis.readUTF());
65                 }else if(ae.getActionCommand().equals("W")){
66                     _dos.writeInt(WEST);
67                     System.out.println(_dis.readUTF());
68                 }else if(ae.getActionCommand().equals("NW")){
69                     _dos.writeInt(NORTH_WEST);
70                     System.out.println(_dis.readUTF());
71                 }
72             }catch(Exception e){
73                 System.out.println("actionPerformed(): problem with socket IO.");
74                 e.printStackTrace();
75             }
76         }
77     }
78
79     public void setUpGui(){

```

```

80     _button1 = new JButton("NW");
81     _button1.addActionListener(this);
82
83     _button2 = new JButton("N");
84     _button2.addActionListener(this);
85
86     _button3 = new JButton("NE");
87     _button3.addActionListener(this);
88
89     _button4 = new JButton("W");
90     _button4.addActionListener(this);
91
92     _button5 = new JButton("");
93     _button5.addActionListener(this);
94
95     _button6 = new JButton("E");
96     _button6.addActionListener(this);
97
98     _button7 = new JButton("SW");
99     _button7.addActionListener(this);
100
101     _button8 = new JButton("S");
102     _button8.addActionListener(this);
103
104     _button9 = new JButton("SE");
105     _button9.addActionListener(this);
106
107     this.getContentPane().setLayout(new GridLayout(3,3,0,0));
108     this.getContentPane().add(_button1);
109     this.getContentPane().add(_button2);
110     this.getContentPane().add(_button3);
111     this.getContentPane().add(_button4);
112     this.getContentPane().add(_button5);
113     this.getContentPane().add(_button6);
114     this.getContentPane().add(_button7);
115     this.getContentPane().add(_button8);
116     this.getContentPane().add(_button9);
117     this.setSize(200, 200);
118     this.setLocation(300, 300);
119     this.pack();
120     this.show();
121 } // end setUpGui() method
122
123 public static void main(String[] args){
124     try{
125         new Socket_NetRatClient(args[0]);
126     }catch(ArrayIndexOutOfBoundsException e1){
127         System.out.println("Usage: java NetRatClient <host>");
128     }
129     catch(Exception e2){
130         e2.printStackTrace();
131     }
132 } // end main() method
133 } // end Socket_NetRatClient class definition

```

Referring to example 20.17—the code for the `Socket_NetRatClient` class looks very similar to its `RMI_NetRatClient` cousin. The similarities between the two lay mostly with the look-and-feel of the user interface. Users send commands to robot rats using a 3 x 3 grid of buttons. However, in this version of the client, commands are sent to the server via a `Socket` connection using the `DataOutputStream` class. Server responses are read using the `DataInputStream` class.

The `Socket` connection is established in the body of the constructor. The host is specified on the command line when the client application is launched. The bulk of the processing takes place in the body of the `actionPerformed()` method starting on line 45. In response to directional button clicks the corresponding command is sent to the server. The server response is immediately expected and processed. In this case the server response string is simply printed to the console.

With the socket-based client application complete you are now ready to test the results of the fourth iteration development activities.

TESTING THE FOURTH ITERATION CODE

You can now have some real fun with the robot rat application. Crank up the NetRatServer then launch several Socket_NetRatClient and RMI_NetRatClient applications. Test from the local machine and from a remote computer if you have one available.

I will not step through the testing of this iteration as I did for the previous one. Socket-based client robot rats look and feel the same on the floor as their RMI-controlled counterparts. There are, however, a few loose ends that need to be addressed before the NetRatServer application can be called complete. The primary issue at hand is that when clients disconnect from the server their corresponding rat images remain on the floor. It would be nice if they would disappear when the client disconnected.

Secondly, several “magic values” appear in the body of the server code. (*i.e.*, *the size of the floor’s JFrame and its location on the screen, the number of pixels the rat image moves, etc.*) From a configuration standpoint magic values render a program difficult to maintain. A change in floor size, for example, would require the RobotRat class to be recompiled, its stub regenerated, and the code redistributed. For a large server application this requirement alone would be a show-stopper.

Finally, code that appears in the application for testing purposes can be removed. This primarily applies to the two instances of RobotRat that are created in the NetRatServer application. These issues are addressed in one last implementation iteration.

Quick Review

Programming a socket-based client-server application requires attention to lower-level details when compared with RMI-based applications. To handle multiple socket-based client connections, a socket-based server application must be multi-threaded. The socket-based server must also establish an application protocol so the client application can effectively communicate with the server application.

NETRATSERVER IMPLEMENTATION – FINAL ITERATION

The NetRatServer application has some cool functionality. You can connect to it and control robot rats with two types of client applications. However, the persistence of the robot rat images on the floor after clients disconnect is a minor irritant, and the presence of magic values in the server code is problematic. This iteration implements a solution to both of these issues. Table 20-8 presents the design considerations and decision for this final NetRatServer implementation iteration.

Check-Off	Design Consideration	Design Decision
	Objectives for final iteration server application development activities.	The objectives for this iteration are three fold: First, implement a mechanism that will remove the rat images from the floor when clients disconnect from the server application. Second, eliminate embedded magic values from the server application code to improve maintainability and make it easy to make server configuration changes. The RobotRat and NetRatServer source file must be examined to identify candidate magic values. Third, test code can be removed from the NetRatServer class.

Table 20-8: Final Iteration Design Considerations and Decisions

Check-Off	Design Consideration	Design Decision
	NetRatServerProperties class	The NetRatServerProperties class will extend the java.util.Properties class. Server configuration values will be stored here and retrieved by server application components as required. Consolidating server configuration values in one location will significantly improve maintainability and make it easier to effect configuration changes. The NetRatServerProperties class will also implement the Singleton pattern. The Singleton pattern is used when you want only one object of a particular class type to exist. Any component within the NetRatServer application can gain access to the single instance of the NetRatServerProperties object to store and retrieve properties as required.
	RobotRat class	The RobotRat class must be modified to utilize the NetRatServerProperties class.
	NetRatServer class	The NetRatServer class must be modified to utilize the NetRatServerProperties class. The code that creates the two instances of RobotRat for testing purposes can be removed. However, when this code is removed the floor will not be drawn on the screen until the first client connects to the server. (This is due to the fact that the RobotRat class is not loaded into the Java Virtual Machine until it is needed by the server application. This may or may not be the desired server behavior but, for this example, the NetRatServer will explicitly load the class on application startup using the Class.forName() method.
	rmic tool	The RobotRat_stub.class file will need to be generated with the rmic tool and deployed to the server side and client side.

Table 20-8: Final Iteration Design Considerations and Decisions

The final NetRatServer application design is shown in figure 20-21.

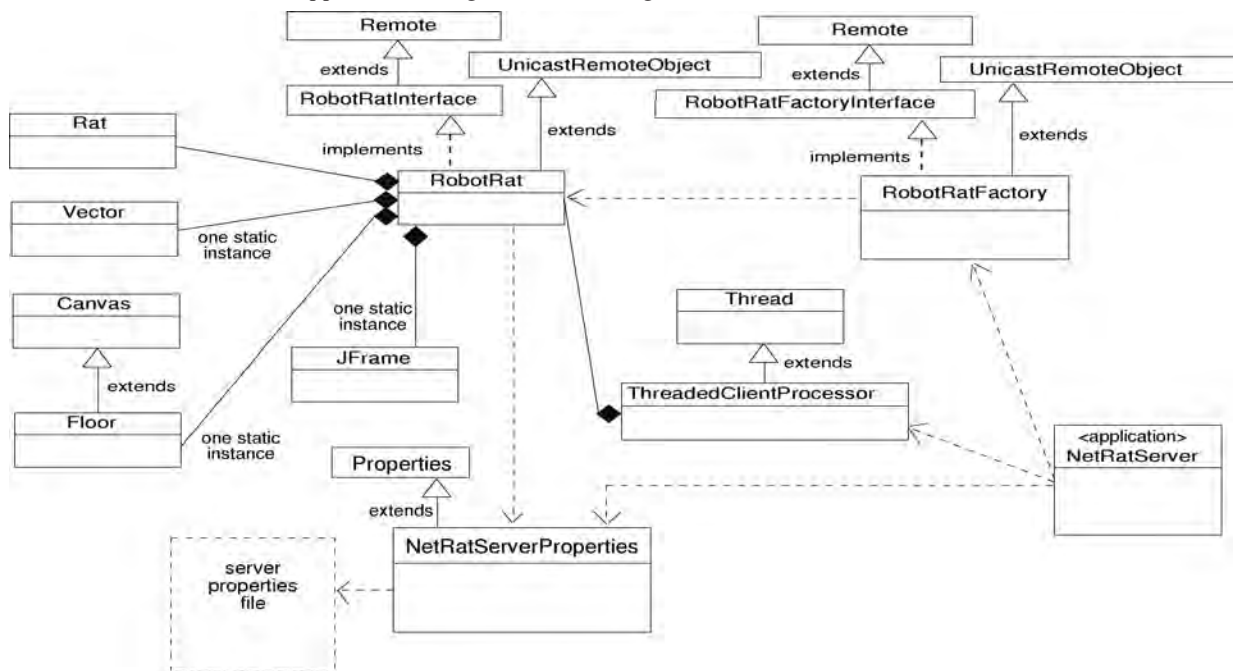


Figure 20-21: Final NetRatServer Application Design Class Diagram

Referring to figure 20-21 — the new class, `NetRatServerProperties`, has been added to the mix. The `RobotRat` and `NetRatServer` classes now have a dependency on the `NetRatServerProperties` class. The `NetRatServerProperties` class stores its properties in a properties file.

Although not shown in the diagram, the `NetRatServerProperties` object implements the Singleton pattern. The singleton pattern is employed when you want only one instance of a particular class type to exist for use within an application.

Another change to the design that is evident in the diagram was made to the `NetRatServer` class. It no longer extends `JFrame` or implements `ActionListener`. The test code previously used to create two instances of `RobotRat` and move them around the floor with the single move button was removed.

FINAL ITERATION CODE

The following code is developed as a result of this iteration.

20.18 NetRatServerProperties.java

```

1      /*****
2      * FileName: NetRatServerProperties.java
3      * @author Rick Miller
4      *
5      * Description: Implements the singleton pattern. A NetRatServerProperties object implements
6      * the persistence of application information required by the NetRatServer application.
7      *****/
8      import java.util.*;
9      import java.io.*;
10
11     /*****
12     * NetRatServerProperties class Description: Implements the singleton pattern. A NetRatServerProperties
13     * object implements the persistence of application information required by the NetRatServer
14     * application.
15     * @see Properties
16     *****/
17
18     public class NetRatServerProperties extends Properties {
19
20         // class constants
21         public static final String DEFAULT_RMI_PORT = "DEFAULT_RMI_PORT";
22         public static final String DEFAULT_SOCKET_PORT = "DEFAULT_SOCKET_PORT";
23         public static final String DEFAULT_SERVER_IP = "DEFAULT_SERVER_IP";
24         public static final String DEFAULT_PROPERTIES_FILE = "DEFAULT_PROPERTIES_FILE";
25         public static final String DEFAULT_PROPERTIES_FILENAME = "netratserver.properties";
26         public static final String DEFAULT_FRAME_HEIGHT = "DEFAULT_FRAME_HEIGHT";
27         public static final String DEFAULT_FRAME_WIDTH = "DEFAULT_FRAME_WIDTH";
28         public static final String DEFAULT_FRAME_X_POSITION = "DEFAULT_FRAME_X_POSITION";
29         public static final String DEFAULT_FRAME_Y_POSITION = "DEFAULT_FRAME_Y_POSITION";
30         public static final String DEFAULT_STEP_SIZE = "DEFAULT_STEP_SIZE";
31         public static final String DEFAULT_ROBOTRAT_CLASS_NAME = "DEFAULT_ROBOTRAT_CLASS_NAME";
32         public static final String DEFAULT_ROBOTRAT_FACTORY_SERVICE_NAME =
33             "DEFAULT_ROBOTRAT_RMI_SERVICE_NAME";
34
35         // class variables
36         private static NetRatServerProperties _properties_object = null;
37
38
39     /*****
40     * Private constructor
41     * Design Decisions:
42     * -- Attempts to open properties file and load persistent properties. If that fails, it will
43     * try to create the properties file, set itself up with default property values, and attempt
44     * to store the property values in the file. If that fails...check your hard drive!
45     *****/
46
47     private NetRatServerProperties( String properties_file ){
48         try{
49             FileInputStream fis = new FileInputStream(properties_file);
50             load(fis);
51         }catch(Exception e) {
52             System.out.println("Problem opening properties file!");
53             System.out.println("Bootstrapping properties...");
54             try{
55                 FileOutputStream fos = new
56                     FileOutputStream(NetRatServerProperties.DEFAULT_PROPERTIES_FILENAME);
57                 setProperty(NetRatServerProperties.DEFAULT_RMI_PORT, "1099");
58                 setProperty(NetRatServerProperties.DEFAULT_SOCKET_PORT, "5001");
59                 setProperty(NetRatServerProperties.DEFAULT_SERVER_IP, "127.0.0.1");
60                 setProperty(NetRatServerProperties.DEFAULT_PROPERTIES_FILE, "netratserver.properties");

```

```

61         setProperty(NetRatServerProperties.DEFAULT_FRAME_HEIGHT, "300");
62         setProperty(NetRatServerProperties.DEFAULT_FRAME_WIDTH, "300");
63         setProperty(NetRatServerProperties.DEFAULT_FRAME_X_POSITION, "200");
64         setProperty(NetRatServerProperties.DEFAULT_FRAME_Y_POSITION, "200");
65         setProperty(NetRatServerProperties.DEFAULT_STEP_SIZE, "3");
66         setProperty(NetRatServerProperties.DEFAULT_ROBOTRAT_CLASS_NAME, "RobotRat");
67         setProperty(NetRatServerProperties.DEFAULT_ROBOTRAT_FACTORY_SERVICE_NAME,
68                     "Robot_Rat_Factory");
69
70         super.store(fos, "NetRatServerProperties File - Edit Carefully");
71         fos.close();
72     } catch (Exception e2) { System.out.println("Uh ohh...Bigger problems exist!"); }
73     }
74 }
75
76
77 /*****
78 * Private default constructor. Applications will get an instance via the getInstance() method.
79 * @see getInstance()
80 *****/
81 private NetRatServerProperties() {
82     this(DEFAULT_PROPERTIES_FILENAME);
83 }
84
85 /*****
86 * The store() method attempts to persist its properties collection.
87 *****/
88 public void store() {
89     try {
90         FileOutputStream fos = new
91             FileOutputStream(getProperty(NetRatServerProperties.DEFAULT_PROPERTIES_FILE));
92         super.store(fos, "NetRatServerProperties File");
93         fos.close();
94     } catch (Exception e) { System.out.println("Trouble storing properties!"); }
95 }
96
97 /*****
98 * getInstance() returns a singleton instance if the NetRatServerProperties object.
99 *****/
100
101 public static NetRatServerProperties getInstance() {
102     if (_properties_object == null) {
103         _properties_object = new NetRatServerProperties();
104     }
105     return _properties_object;
106 }
107 } // end NetRatServerProperties class definition

```

20.19 RobotRat.java

```

1  import javax.swing.*;
2  import java.util.*;
3  import java.rmi.*;
4  import java.rmi.server.*;
5  import java.awt.*;
6
7  public class RobotRat extends UnicastRemoteObject implements RobotRatInterface {
8
9      /* *****
10     static variables
11     *****/
12     private static JFrame _frame = null;
13     private static Floor _floor = null;
14     private static Vector _rats = null;
15     private static int _frame_height;
16     private static int _frame_width;
17     private static int _frame_x_position;
18     private static int _frame_y_position;
19     private static NetRatServerProperties _properties = null;
20
21     /* *****
22     instance variables
23     *****/
24     private Rat _its_rat = null;
25     private int _step_size;
26
27
28     static {
29         _properties = NetRatServerProperties.getInstance();
30         _rats = new Vector();
31         _floor = new Floor(_rats);

```

```

32     _frame = new JFrame("Rat Movement Floor");
33     _frame.getContentPane().add(_floor);
34     _frame_height =
35         Integer.parseInt(_properties.getProperty(NetRatServerProperties.DEFAULT_FRAME_HEIGHT));
36     _frame_width =
37         Integer.parseInt(_properties.getProperty(NetRatServerProperties.DEFAULT_FRAME_WIDTH));
38     _frame.setSize(_frame_height, _frame_width);
39     _frame_x_position =
40         Integer.parseInt(_properties.getProperty(NetRatServerProperties.DEFAULT_FRAME_X_POSITION));
41     _frame_y_position =
42         Integer.parseInt(_properties.getProperty(NetRatServerProperties.DEFAULT_FRAME_Y_POSITION));
43     _frame.setLocation(_frame_x_position, _frame_y_position);
44     _frame.show();
45 }
46
47
48 public RobotRat() throws RemoteException {
49     _its_rat = new Rat();
50     _rats.addElement(_its_rat);
51     _step_size = Integer.parseInt(_properties.getProperty(NetRatServerProperties.DEFAULT_STEP_SIZE));
52     _floor.repaint();
53 }
54
55 protected void finalize() throws Throwable {
56     System.out.println("RobotRat finalize() called!");
57     if(_its_rat != null){
58         _rats.removeElement(_its_rat);
59         _its_rat = null;
60     }
61 }
62
63
64 public void moveEast(){
65     _its_rat.setX(_its_rat.getX() + _step_size);
66     _floor.repaint();
67 }
68
69 public void moveSouth(){
70     _its_rat.setY(_its_rat.getY() + _step_size);
71     _floor.repaint();
72 }
73
74 public void moveWest(){
75     _its_rat.setX(_its_rat.getX() - _step_size);
76     _floor.repaint();
77 }
78
79 public void moveNorth(){
80     _its_rat.setY(_its_rat.getY() - _step_size);
81     _floor.repaint();
82 }
83
84 public void moveNorthWest(){
85     _its_rat.setY(_its_rat.getY() - _step_size);
86     _its_rat.setX(_its_rat.getX() - _step_size);
87     _floor.repaint();
88 }
89
90 public void moveSouthWest(){
91     _its_rat.setY(_its_rat.getY() + _step_size);
92     _its_rat.setX(_its_rat.getX() - _step_size);
93     _floor.repaint();
94 }
95
96 public void moveNorthEast(){
97     _its_rat.setY(_its_rat.getY() - _step_size);
98     _its_rat.setX(_its_rat.getX() + _step_size);
99     _floor.repaint();
100 }
101
102 public void moveSouthEast(){
103     _its_rat.setY(_its_rat.getY() + _step_size);
104     _its_rat.setX(_its_rat.getX() + _step_size);
105     _floor.repaint();
106 }
107 } // end RobotRat class definition

```


20.20 NetRatServer.java

```

1      import java.rmi.*;
2      import java.rmi.registry.*;
3      import java.net.*;
4
5      public class NetRatServer {
6          public static void main(String[] args){
7              NetRatServerProperties properties = NetRatServerProperties.getInstance();
8              System.out.println("NetRatServer Lives!!");
9              try{
10                 Class.forName(properties.getProperty(NetRatServerProperties.DEFAULT_ROBOTRAT_CLASS_NAME));
11                 System.out.println("Starting registry...");
12                 LocateRegistry.createRegistry(Integer.parseInt(
13                     properties.getProperty(NetRatServerProperties.DEFAULT_RMI_PORT));
14                 System.out.println("Registry started on port " +
15                     properties.getProperty(NetRatServerProperties.DEFAULT_RMI_PORT) + ".");
16                 System.out.println("Binding service name to remote object...");
17                 Naming.bind(properties.getProperty(
18                     NetRatServerProperties.DEFAULT_ROBOTRAT_FACTORY_SERVICE_NAME), new RobotRatFactory());
19                 System.out.println("Bind successful.");
20                 System.out.println("Ready for remote method invocation.");
21
22                 System.out.println("Creating ServerSocket Object...");
23                 ServerSocket server_socket = new ServerSocket(Integer.parseInt(properties.getProperty(
24                     NetRatServerProperties.DEFAULT_SOCKET_PORT));
25                 System.out.println("ServerSocket object created successfully!");
26                 while(true){
27                     System.out.println("Listening for incoming client connections...");
28                     Socket socket = server_socket.accept();
29                     System.out.println("Incoming client connection detected.");
30                     System.out.println("Creating ThreadedClientProcessor object...");
31                     ThreadedClientProcessor client_processor = new ThreadedClientProcessor(socket);
32                     System.out.println("ThreadedClientProcessor object created.");
33                     System.out.println("Calling start() method...");
34                     client_processor.start();
35                 }
36             }catch(Exception e){
37                 e.printStackTrace();
38             }
39         }
40     } // end NetRatServer class

```

Referring to example 20.18 through 20.20 —The `NetRatServerProperties` class extends the `java.util.Properties` class. Lines 21 through 33 declare a set of class-wide `String` constants that represent the keys through which property values will be accessed. The actual default property values are set in the body of the constructor.

When an instance of `NetRatServerProperties` is created it will attempt to load the `netratserver.properties` file. If the properties file cannot be located a new copy will be created which contains the required default values. Once the `netratserver.properties` file exists, changes to the server application configuration can be made to this file where appropriate before the server application is started. In this manner application behavior can be changed without recompiling the source code.

The `RobotRat` and `NetRatServer` classes have been modified to utilize the `NetRatServerProperties` class. In each case the Singleton instance of `NetRatServerProperties` is retrieved via the static `getInstance()` method. (See line 29 of example 20-19 or line 7 of example 20.20.) Property values are retrieved from the properties object as required via the `getProperty()` method. Note that all property values are stored in the properties object as `Strings`. In cases where an integer value is expected in the code, say to set the size of the `JFrame` in the `RobotRat` class, the `String` value must be converted to an integer value via the `Integer.parseInt()` method.

Notice the use of the `Class.forName()` method on line 10 of example 20.20. The `RobotRat` class is explicitly loaded so that the floor displays when the server starts up. The name of the `RobotRat` class is also retrieved from the properties object.

The `RobotRat` class has also been modified to include a `finalize()` method starting on line 55 of example 20.19. The `finalize` method is called when the Java virtual machine garbage collector collects an unreferenced `RobotRat` object. This will result in the `_its_rat` reference being set to null which will effectively remove the rat image from the floor. However, when exactly this will occur is purely at the discretion of the virtual machine garbage collector. A `RobotRat` object created for a socket-based client will be collected sooner when the client disconnects than will a `RobotRat` object created for an RMI-based client. This is due to the Java RMI runtime holding a reference to the `RobotRat` object for quite some time after the RMI-based client disconnects. Testing on my computer resulted in the RMI runtime holding on to `RobotRat` objects for as long as ten minutes. In any case, you cannot guarantee when,

exactly, the `finalize()` method will be called, and you cannot call it explicitly in any case. Rat images will, for now, have to linger upon the floor until the garbage collector removes their corresponding `RobotRat` objects from memory.

TESTING THE FINAL ITERATION CODE

When the `NetRatServer` application is run for the first time after making the required modifications the `netratserver.properties` file will not exist. Figure 20-22 shows the console output when the `NetRatServer` application is started and the `netratserver.properties` file cannot be located.

Notice the properties file must be created if it does not exist.



```

Terminal — java
[Rick-Millens-Computer:~] swadog% cd /Users/swadog/Desktop/NetRatServer/Final_Iteration
[Rick-Millens-Computer:~/Desktop/NetRatServer/Final_Iteration] swadog% java NetRatServer
Problem opening properties file!
Bootstrapping properties...
NetRatServer Lives!
Starting registry...
Registry started on port 1099.
Binding service name to remote object...
RobotRatFactoryImplementation object created!
Bind successful...
Ready for remote method invocation...
Creating ServerSocket Object...
ServerSocket object created successfully!
Listening for incoming client connections...

```

Figure 20-22: Console Output on `NetRatServer` Application Startup

Figure 20-23 shows the empty floor waiting for the arrival of the first robot rat image. The floor is displayed as a result of explicitly loading the `RobotRat` class using the `Class.forName()` method. Start up a few socket-based and



Figure 20-23: Empty Floor Displayed as a Result of Explicitly Loading the `RobotRat` Class

RMI-based clients and move the rats around the floor. Shutdown the clients and note how long it takes for the JVM on your computer to collect the `RobotRat` objects.

PARTING COMMENTS

The robot rat client-server application presented in this chapter has some interesting functionality and demonstrates many important concepts regarding network-based client-server application design. Space limitations of this book force me to bring the discussion of this project to a close. However, thorough study of the techniques and code presented in this chapter combined with a little imagination will enable you to expand the scope of this project and make it a really neat application while having fun at the same time. Suggestions for improvements to the robot rat client-server application, as well as ideas for other client-server projects, are offered in the `Skill-Building` and `Suggested Projects` sections.

Quick Review

Hard-coded magic values render it difficult to change an application's configuration settings. The `Properties` class can be used to store and retrieve application property values.

There is no guarantee when the JVM garbage collector will run and remove unreferenced objects from memory. Objects referenced by the RMI runtime will generally be held longer than objects referenced by independent threads.

SUMMARY

Simple socket-based server applications utilize a `ServerSocket` object to listen for incoming client connections. A `ServerSocket` object listens for incoming connections on a specified port. Client applications utilize a `Socket` object to connect to a server application on a specified URL or IP address and port number. The `localhost` IP address can be used during testing.

The `Socket` object is used to create the `InputStream` objects using the `getInputStream()` and `getOutputStream()` methods. Once the `InputStream` objects are created on both the client and server ends the applications can communicate with each other via an established set of rules. (*a.k.a., custom application protocol*). A client-server protocol is connection oriented if a socket connection is maintained open between the client and server applications for the duration of their communication session.

Proceed first with a client-server project by giving sufficient analysis to the project specification. Use noun-verb analysis to discover candidate application components and actions. Sketch out a rough application design to use as an implementation road map. Make a list of candidate application classes and assign to them an initial set of responsibilities. The objective of the analysis and design phase is to provide a starting point for implementation and get the creative juices flowing.

Socket-based client-server applications will utilize classes found in the `java.net` package. RMI-based client-server applications will utilize classes found in the `java.rmi` package and its related packages. However, to fully implement a client-server application requires the utilization of many different Java platform classes.

When ready to proceed with the implementation phase select the smallest subset of the design that results in a set of application functionality that can be tested. Some code written during the implementation may be ultimately discarded before the final application is released.

Remote Method Invocation functionality is often easier to implement than socket-based functionality. This is because the RMI runtime handles socket communication and thread management issues behind the scenes. Implementing RMI functionality, however, requires the programmer to generate the necessary stub files for deployment with the server and client applications. RMI servers must bind an instance of the RMI object to a service name in an RMI registry.

RMI-based clients gain access to an RMI object via its service name. The remote object's interface and stub classes must be deployed with the client application. Calls to the remote object look like ordinary method calls. All network communication required between the RMI-based client and remote object is handled by the RMI runtime environment.

The RMI runtime manages client access to RMI server-side objects. In cases where RMI-based clients must have access to dedicated server-side remote objects a factory class can be employed to create the dedicated remote objects as required. RMI-based clients must first get a reference to the server-side factory object and use it to create the dedicated remote object it requires. This solves the dedicated remote object problem but raises the level of application complexity. In addition to the dedicated remote object interface and stub classes, the factory interface and stub classes must also be deployed with the server and client applications.

Programming a socket-based client-server application requires attention to lower-level details when compared with RMI-based applications. To handle multiple socket-based client connections, a socket-based server application must be multi-threaded. The socket-based server must also establish an application protocol so the client application can effectively communicate with the server application.

Hard-coded magic values render it difficult to change an application's configuration settings. The `Properties` class can be used to store and retrieve application property values.

There is no guarantee when the JVM garbage collector will run and remove unreferenced object from memory. Objects referenced by the RMI runtime will generally be held longer than objects referenced by independent threads.

Skill-Building Exercises

1. **Programming:** Modify the simple socket-based server application given in example 20.1 so that it is multi-threaded. You can get some ideas on how to proceed with this exercise by studying the `ThreadedClientProcessor` class code given in example 20.15. Test your work by attempting to connect with multiple `SimpleClient` applications.
2. **Programming:** Modify the robot rat client-server application presented in this chapter so that clients have some way of keeping track of their robot rat's location on the floor. One possible solution might be to give the clients their own local floor. This modification will require changes to both the client and the server applications. The server will need to return rat position data to the client after each move. The client application must be able to properly utilize this data in some way.
3. **Programming:** Modify the robot rat client-server application presented in this chapter so that clients can see the positions on the floor of all the other robot rats, including its own.
4. **Programming:** Modify the robot rat client-server application presented in this chapter so that clients can destroy enemy robot rats.
5. **Programming:** Modify the robot rat client-server application presented in this chapter so that the server application configuration settings can be set via a dialog window from a client application. Make the ability to change the server setting password controlled.
6. **Programming:** Modify the robot rat client-server application presented in this chapter so that the height and width of the floor, and its location on the computer screen, will be automatically saved into the `netratserver.properties` file whenever the floor is moved or resized. The following example code will give you a clue regarding how you might proceed with this enhancement.

20.21 *PCTest.java*

```

1      import java.awt.event.*;
2      import java.awt.*;
3      import javax.swing.*;
4
5      public class PCTest extends JFrame implements ComponentListener {
6
7          public PCTest(){
8              this.setSize(200, 200);
9              this.setLocation(300, 300);
10             this.addComponentListener(this);
11             this.show();
12         }
13
14         public void componentHidden(ComponentEvent e){ }
15         public void componentMoved(ComponentEvent e){
16             Point point = this.getLocation(new Point());
17             System.out.println("Location x: " + point.getX() + " " + "Location y: " + point.getY());
18         }
19
20         public void componentResized(ComponentEvent e){
21             System.out.println("Height: " + this.getHeight() + " " + "Width: " + this.getWidth());
22         }
23
24         public void componentShown(ComponentEvent e){ }
25
26         public static void main(String[] args){
27             new PCTest();
28         }
29     } // end PCTest class definition
30

```

- 7. Programming:** Modify the robot rat client-server application presented in this chapter so that client applications have the ability to add or remove robot rats. Ensure that all of a client's remaining robot rats are removed from the floor when a client application disconnects before removing all of its robot rats. Also, give the client applications the ability to select different robot rats for movement or removal.
- 8. Programming:** Modify the robot rat client-server application presented in this chapter so that robot rat images displayed on the floor include some kind of client ID that allows robot rats to be correlated with their owning client application.

SUGGESTED PROJECTS

- 1. Create Naval Warfare Game:** Create a client-server application that allows multiple socket-based client applications to conduct naval warfare against each other. Each client can add ships to the "ocean" surface at a specified location. Each client can fire a shot blindly into any spot in the ocean. If their shot hits an opponent's ship that ship is sunk and removed from the ocean surface.
- 2. Remotely Access Legacy Datafile:** Using the `DataFileAdapter` class presented in chapter 18, example 18.23, create a multi-threaded server application that allows socket-based client applications to remotely query and update the data file.
- 3. Remotely Access Legacy Datafile:** Using the `DataFileAdapter` class presented in chapter 18, example 18.23, create an RMI-based server application that allows RMI-based client applications to remotely query and update the data file.
- 4. Chat Program:** Write a client-server application that allows multiple socket- or RMI-based clients to connect to a server and exchange messages with other connected users.
- 5. Secure Chat Program:** Modify the chat program created in project 4 so that users have the option of sending messages in the clear (unencrypted) or encrypted.
- 6. Remote Application Launcher:** Write a client-server application that allows client applications to browse a selection of server-side Java applications, select one, and launch it while observing its console output in a client-side text area.

SELF-TEST QUESTIONS

1. List and describe each step of the socket-based client-server connection scenario.
2. What class from the `java.net` package is used to listen for incoming socket connections?
3. Describe in general terms what is required to create a multi-threaded server application.
4. List and describe the steps involved with creating an RMI-based server application.
5. What must an RMI-based client do before it can use the services of a remote object?
6. What must be established prior to conducting effective socket-based client-server communications?
7. What interface does a remote object's interface extend?

8. What class from the `java.rmi.server` package does a remote object extend? What interface does it implement?
9. What method found in the `ServerSocket` class blocks until an incoming client connection is detected?
10. What is the default RMI registry port number?
11. If a UTF string is written to an output stream using the `DataOutputStream.writeUTF()` method, what method of what class must be called to read the string on the receiving end?
12. What methods of the `java.net.Socket` class are used to retrieve the `InputStream` and `OutputStream` objects from the socket?

REFERENCES

Jim Farley, et. al. *Java Enterprise In A Nutshell: A Desktop Quick Reference*, Second Edition. O'Reilly & Associates, Sebastopol, CA. ISBN: 0-596-00152-5

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*, 4th Edition. O'Reilly and Associates, Inc. Sebastopol, CA. ISBN: 0-596-00283-1

The Java 2 Standard Edition Version 5 Platform API Online Documentation [<http://java.sun.com/j2se/1.5.0/docs/api/index.html>]

The Java 2 Standard Edition Version 1.4.2 Platform API Online Documentation [<http://java.sun.com/j2se/1.4.2/docs/api/index.html>]

Merlin Hughes, et. al. *Java Network Programming, Second Edition*. Manning Publications Company, Greenwich, CT. ISBN: 1-884777-49-X

William Grosso. *RMI, Dynamic Proxies, and the Evolution of Deployment*. [<http://today.java.net/pub/a/today/2004/06/01/RMI.html>]

Java RMI Release Notes for J2SE 5.0 [<http://java.sun.com/j2se/1.5.0/docs/guide/rmi/relnotes.html>]

Java RMI Release Notes for J2SE 1.4.2 [<http://java.sun.com/j2se/1.4.2/docs/guide/rmi/relnotes.html>]

NOTES

CHAPTER 21



Auntie Liz

Applets & JDBC

LEARNING OBJECTIVES

- *STATE THE PURPOSE AND USE OF APPLETS*
- *LIST AND DESCRIBE THE APPLET LIFE-CYCLE STAGES*
- *LIST AND DISCUSS THE PRIMARY ISSUES INVOLVED WHEN DESIGNING WITH APPLETS*
- *LIST AND DESCRIBE APPLET SECURITY RESTRICTIONS*
- *USE THE <APPLET> TAG TO EMBED APPLETS IN HTML PAGES*
- *UTILIZE APPLET PARAMETERS TO PASS INFORMATION INTO AN APPLET FROM AN HTML PAGE*
- *WRITE AN APPLET THAT UTILIZES REMOTE METHOD INVOCATION TO SEND DATA TO A SERVER FOR PERSISTENCE INTO A RELATIONAL DATABASE*
- *UTILIZE SWING COMPONENTS IN AN APPLET*
- *UTILIZE JDBC TO QUERY, INSERT, AND UPDATE DATA IN A RELATIONAL DATABASE*
- *MANIPULATE RESULTSETS AND RESULTSET METADATA IN JDBC PROGRAMS*
- *UTILIZE CLASSES FOUND IN THE java.sql PACKAGE TO WRITE JDBC PROGRAMS*
- *CREATE AND EXECUTE PREPARED STATEMENTS IN JDBC PROGRAMS*

INTRODUCTION

The primary focus of this chapter is on the topic of Java applets. An applet is a Java program that can be embedded in an HTML page and run in a web browser. An applet can be as feature-rich as an ordinary Java application although the security restrictions placed on an applet put a practical limit on its functionality. Here you will learn about an applet's life cycle stages and how to embed applets in HTML pages using the `<applet>` tag. You will also learn about applet security restrictions and how to approach applet program design with these security restrictions in mind.

The secondary focus of this chapter is on writing Java programs that can manipulate relational databases using the JDBC API. The comprehensive project in this chapter will show you how to write an applet that connects to, and exchanges data with, a server via Remote Method Invocation (RMI). The server application will translate the RMI method calls into JDBC calls against a MySQL relational database.

I attempt to cover a lot of ground in this chapter and in the interest of keeping the page count within reason I'm going to cry uncle up front and tell you what I'm not going to talk about. I will not dive deeply into relational database design or the Structured Query Language (SQL). Volumes have been written about these subjects and their complete treatment is simply not possible in one chapter. I will also not dwell too long on the topic of MySQL other than to say that it is a great open source database and to provide some guidance on how to get it up and running as quickly as possible. I discuss how to get it, install it, and fiddle with it a bit to get it to work for you. (*I use the term "fiddle with it" in the affectionate sense.*) I have listed several good references related to all three topics at the end of the chapter.

The issues you must resolve regardless of what database product you use are the same. You must obtain it, install it, configure it (*create databases, tables, users, permissions, etc.*), determine the name and location of the appropriate JDBC driver class, and formulate the correct database connection URL. Getting the database installed, configured, and ready to go represent easily 90% of the challenges you will encounter in trying to learn and understand the peripheral concepts related to the material presented in this chapter. If you are a novice, learning Java, SQL, JDBC, and the idiosyncrasies of your preferred database all at once can be overwhelming. I will make every effort to focus your attention on the important points regarding these issues so you can move forward with confidence in learning about applets and JDBC.

APPLET OVERVIEW

An applet is a Java program that can be embedded in an HTML page with an `<applet>` or `<object>` tag and hosted on a web server. When a user accesses an HTML page that contains an embedded applet, the applet code is downloaded to the user's machine and the applet is executed within the web browser's Java Virtual Machine or Java Plug-In. Applet code accessed and executed in this fashion is generally considered to be untrusted code, and as such is run under tight security restrictions.

An applet can also be run locally either in a browser or with the help of a special application called an Applet-Viewer. Security restrictions are eased somewhat for locally-run applets but their capabilities are still limited when compared with normal Java applications.

THE BENEFITS OF USING APPLETS

In spite of increased security restrictions and the resulting reduced functionality, the use of applets offers major benefits. The primary benefit derived from using applets is ease of distribution. Normal Java applications must be installed on each client machine. They can be downloaded from a host computer, for sure, but you are assuming your user-base is computer savvy. Also, any changes to the client code requires another round of distribution.

Applets, on the other hand, can be served from a web page. Updates to an applet are posted in one location and the next time a user accesses that page they will use the updated program.

<APPLET> Vs. <OBJECT> TAGS

The <applet> tag is deprecated in the HTML 4.0 specification, superseded by the <object> tag. However, in the interest of backward compatibility, browser creators are enjoined to implement the <applet> tag and most still do. In this chapter I will only demonstrate the use of the <applet> tag. Use of the <object> tag is left as an exercise at the end of the chapter.

BROWSER JVM Vs. JAVA PLUG-IN

There are many different types of browsers in use on the internet at any given instant, some new, many old. Sun provided a technology that enabled applets to run with newer versions of a Java Virtual Machine than what was originally available when the browser was released. This technology is referred to as the Java Plug-In. For more information about the Java Plug-In refer to the following link: [<http://java.sun.com/products/plugin/>]

A Basic Applet Example

This section presents an example applet program for the purposes of discussing its structure and operational life-cycle. The name of the applet is BasicApplet. An applet extends the JApplet class as is shown in figure 21-1.

As you can see, BasicApplet already has a lot of functionality. It is a JApplet, an Applet, a Panel, a Container, a Component, and, like all reference types, an Object. The Applet class provides, among other things, methods to ease the handling of audio clips and images. The JApplet class is the swing version of Applet and provides enhanced swing component handling features including the capability to add menus to an applet.

APPLET CODE

The code for BasicApplet is given in example 21.1 below.

21.1 BasicApplet.java

```

1      import java.applet.*;
2      import java.awt.*;
3      import javax.swing.*;
4
5      public class BasicApplet extends JApplet {
6
7          private static JTextArea textarea = null;
8          private static JScrollPane scrollpane = null;
9
10         public void init(){
11             textarea = new JTextArea(10, 30);
12             scrollpane = new JScrollPane(textarea);
13             getContentPane().add(scrollpane);
14             textarea.append("BasicApplet init() method called.\n");
15             System.out.println("BasicApplet init() method called.");
16         }
17
18         public void start(){
19             System.out.println("BasicApplet start() method called.");
20             textarea.append("BasicApplet start() method called.\n");
21         }
22
23         public void stop(){
24             System.out.println("BasicApplet stop() method called.");
25             textarea.append("BasicApplet stop() method called.\n");
26         }
27
28         public void destroy(){
29             System.out.println("BasicApplet destroy() method called.");
30             textarea.append("BasicApplet destroy() method called.\n");
31         }
32     } // end BasicApplet class definition

```

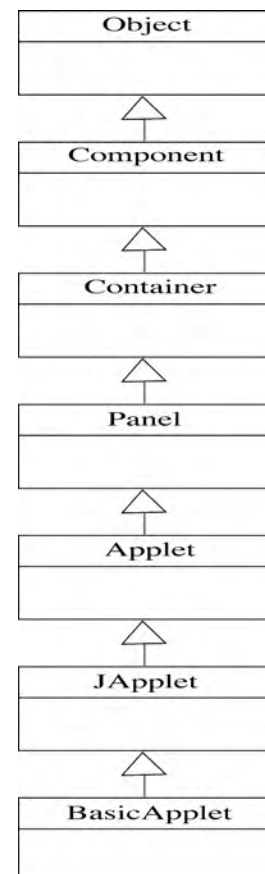


Figure 21-1: BasicApplet Inheritance Hierarchy

Referring to example 21.1 — `BasicApplet` extends `JApplet` and implements the four primary applet methods, `init()`, `start()`, `stop()`, and `destroy()`. These four methods represent milestones in an applet's life cycle. I discuss these life-cycle methods in greater detail below in the section titled *Applet Life-Cycle Stages*.

`BasicApplet` contains a `JTextArea` and a `JScrollPane`. Lines 7 and 8 contain the declarations for the text area and scroll pane references. Both are initialized in the body of the `init()` method. The `init()`, `start()`, `stop()`, and `destroy()` methods append a line of text to the text area as well as print out a short message to the console stating the name of the life-cycle method called. Compile this code as you would a regular Java application, however, before you can run the program you must create an HTML page and use the `<applet>` tag to embed information about the applet you wish to run.

HTML PAGE CODE

Example 21.2 gives the HTML code for a minimal web page that will allow you to run the `BasicApplet` program.

```

1      <html>
2
3      <applet code="BasicApplet.class" height=200 width=300
4          Your browser does not support applets if you are reading this message.
5      </applet>
6
7      </html>

```

21.2 *basicapplet.html*

Referring to example 21.2 — the `<applet>` tag embeds information about an applet in an HTML page. The `<applet>` tag has several attributes, three of which are utilized in this example: `code`, `height`, and `width`. The `code` attribute identifies the name of the applet class file to execute. The `height` and `width` attributes set the height and width of the applet display area. The line of text on line 4 will display if a browser does not recognize the `<applet>` tag.

PACKAGING AND DISTRIBUTION

Before you can run the `BasicApplet` program you must package it with the HTML page. You can do this simply by creating a directory into which you co-locate the `BasicApplet.class` and `basicapplet.html` files.

RUNNING THE BASIC APPLET EXAMPLE

You can test the `BasicApplet` program in several ways. You can run it by opening the `basicapplet.html` file using your web browser. If you do this you are running the applet locally. If you have web-server software installed on your computer you could deploy the directory containing the `BasicApplet` files to your web server and test it locally or via a remote computer. Figure 21-2 shows the `BasicApplet` program running in a web browser in the Apple OS X environment. Figure 21-3 shows the console log and the messages that result from initializing and starting `BasicApplet`.

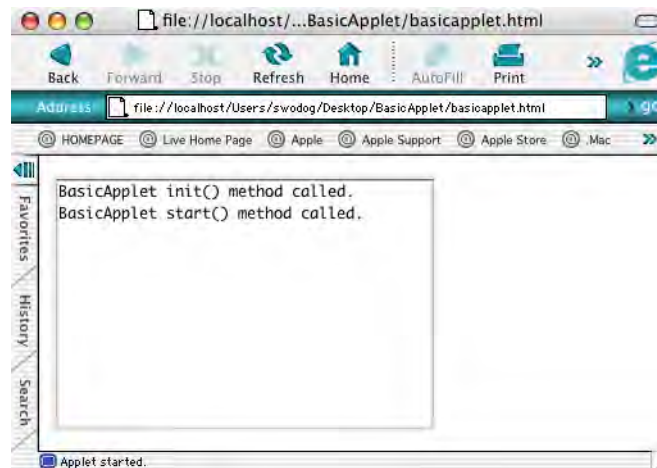


Figure 21-2: `BasicApplet` Running In Web Browser



Figure 21-3: Console Log Showing BasicApplet Life Cycle Messages

Referring to figures 21-2 and 21-3 — When the browser loads the `basicapplet.html` page the `<applet>` tag directs it to load and execute the `BasicApplet.class` file. This results in a call to the applet's `init()` and `start()` methods.

Clicking the browser's back button, refresh button, or shutting down the browser program, results in a call to the applet's `stop()` and `destroy()` methods as is shown in figure 21-4.



Figure 21-4: Console Log Showing BasicApplet Life Cycle Messages After Browser Shuts Down

Applet Life-Cycle Stages

Applets have four life cycle stages: initialized, started, stopped, and destroyed. These stages correspond with the applet methods `init()`, `start()`, `stop()`, and `destroy()` respectively. Applets are not required to implement all four methods and in fact many do not. You will most likely need to implement only the `init()` method at a minimum. Whether or not you need to implement the remaining life-cycle methods depends on what your applet is doing.

Figure 21-5 illustrates the stages of the applet life cycle. Each stage of the applet life cycle is discussed in detail below.

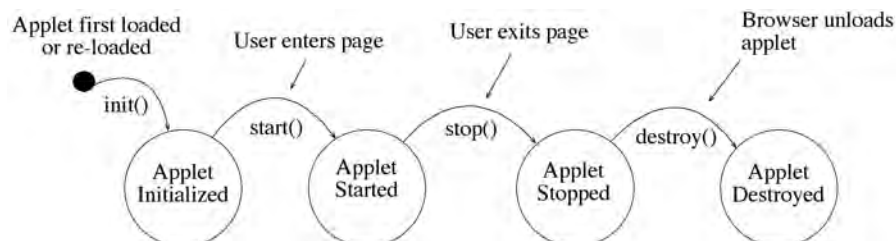


Figure 21-5: Applet Life Cycle Stages

init()

The `init()` method is called when the applet is first loaded or re-loaded by the browser. The `init()` method is akin to a constructor method although an applet can have constructor methods as well. If an applet contains constructor methods, one of them must be a no-argument constructor.

start()

The `start()` method is called after the `init()` method when an applet has been loaded by a browser.

stop()

The `stop()` method is called when the browser exits the page containing the applet or when the browser unloads the applet class. The `stop()` method is called before the `destroy()` method.

destroy()

The `destroy()` method is called when the browser unloads the applet class. The `destroy()` method is a good place to put code that releases any resources retained by the applet such as shutting down socket connections, etc.

THE <APPLET> TAG IN DETAIL

Figure 21-6 shows the `<applet>` tag along with its mandatory and optional attributes. Optional attributes are shown in brackets.

```
<applet code = applet_class_name
        width = width_in_pixels
        height = height_in_pixels
        [archive = jar_file_name, [jar_file_name], ...]
        [codebase = base_applet_uri]
        [object = serialized_applet]
        [alt = alternative_text]
        [name = applet_name]
        [align = applet_alignment]
        [vspace = vertical_margin_space_in_pixels]
        [hspace = horizontal_margin_space_in_pixels] >

[<param name = parameter_name value = parameter_value>]
    ...
[alternative_text]
</applet>
```

Figure 21-6: The `<applet>` Tag and its Attributes

Referring to figure 21-6 — you've seen the `code`, `height`, and `width` `<applet>` tag attributes in action in the BasicApplet example. The remaining attributes are largely self-explanatory although several require some amplification.

The *archive* attribute specifies the name, or list of names, of Java Archive (JAR) files that contain the applet-related program files. The jar files specified will be pre-loaded by the browser. Pre-loading can speed applet execution especially when the applet in question is complex and depends on many files. Large applet programs packaged as jar files download faster because the contents of the jar file are compressed. (*Just like a zip file.*)

The *codebase* attribute specifies the location, in the form of a Uniform Resource Identifier (URI), of the applet class or jar files. You would use the *codebase* attribute if you wanted to locate the applet class or jar files in a location different from that of the applet's HTML page.

The *object* attribute is used if the applet is to be loaded by deserializing a previously serialized applet. You can use either the *code* or the *object* attribute but not both at the same time.

You can set parameter values for use in an applet via `<param>` tags. The use of applet parameters is shown in the section titled *Using Applet Parameters*.

Quick Review

An applet is a Java program that can be embedded in a web page with an `<applet>` tag and run in a web browser. An applet extends the `JApplet` class and thus gains an enormous amount of functionality. Applets have methods that make it easy to manipulate sound and image files.

Applets have four life-cycle stages: initialized, started, stopped, and destroyed. Each stage calls the corresponding life cycle method: `init()`, `start()`, `stop()`, and `destroy()`. Applets do not have to implement all life-cycle methods.

The `init()` method functions much like a constructor and is where applet state initialization code is usually placed. An applet can also have constructor methods if required, one of which must be a no-argument constructor.

Place clean-up code in the `destroy()` method.

Applet Security Restrictions

Applets are considered to be untrusted code, and as such they are subject to tight security restrictions that limit their access to system resources. You can assume that an applet will not have access to the local file system and will not be able to open, read, write, or access files or directories in any way. Additionally, an applet cannot initiate a network connection to or from any computer other than the one from which it was loaded. Figure 21-7 illustrates this point.

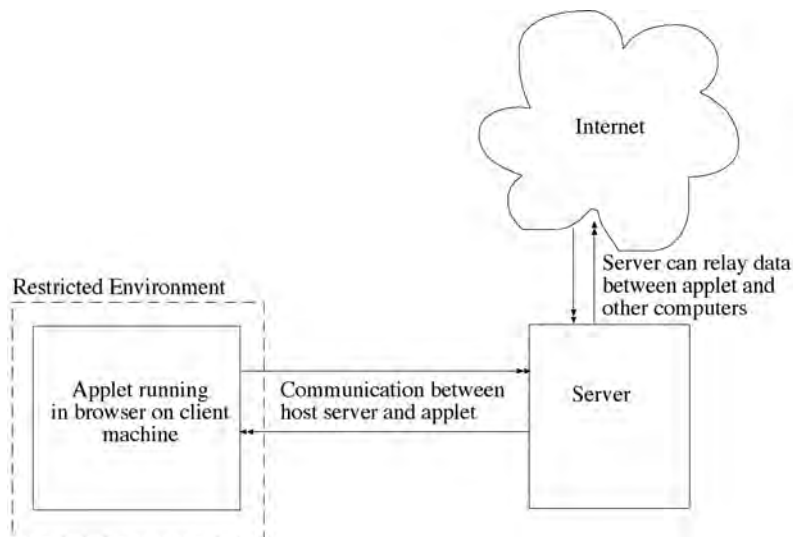


Figure 21-7: An Applet Can Only Connect To The Server From Which It Was Served

Referring to figure 21-7 — The applet executes in a restricted environment on the client machine and can only receive or initiate network connections to the server. However, the server can act as a proxy and communicate with other internet hosts on the applet's behalf.

Any attempt to connect to the applet from a computer other than its serving host will result in an `AccessControlException`. To demonstrate this point examples 21.3 through 21.6 implement an applet named `AppletServer` that mimics the functionality of the `SimpleServer` program originally given in chapter 20.

21.3 *AppletServer.java*

```

1      import java.applet.*;
2      import javax.swing.*;
3
4      public class AppletServer extends JApplet {
5
6          JTextArea _textarea = null;

```

```

7     JScrollPane _scrollpane = null;
8     ServerThread _st = null;
9
10    public void init(){
11        _textarea = new JTextArea(10, 20);
12        _scrollpane = new JScrollPane(_textarea);
13        this.getContentPane().add(_scrollpane);
14        System.out.println("AppletServer init() method called...");
15    }
16
17    public void start(){
18        _st = new ServerThread(_textarea);
19        _st.start();
20        System.out.println("AppletServer start() method called...");
21    }
22 } // end AppletServer class definition

```

21.4 ServerThread.java

```

1     import java.io.*;
2     import java.net.*;
3     import javax.swing.*;
4     import java.security.*;
5
6     public class ServerThread extends Thread {
7
8         private ServerSocket _server_socket = null;
9         private JTextArea _textarea = null;
10
11        public ServerThread(JTextArea text_area){
12            _textarea = text_area;
13            try{
14                _server_socket = new ServerSocket(5001);
15            }catch(Exception e){
16                e.printStackTrace();
17            }
18        } // end constructor
19
20        public void run(){
21            while(true){
22                try{
23                    Socket socket = _server_socket.accept();
24                    ClientProcessorThread cpt = new ClientProcessorThread(socket, _textarea);
25                    cpt.start();
26                }catch(AccessControlException ace){
27                    _textarea.append("Unauthorized network access detected.\n");
28                }
29                catch(Exception e){
30                    e.printStackTrace();
31                }
32            } //end while
33        } // end run() method
34    } // end class definition

```

21.5 ClientProcessorThread.java

```

1     import java.io.*;
2     import java.net.*;
3     import javax.swing.*;
4
5     public class ClientProcessorThread extends Thread {
6
7         private Socket _socket = null;
8         private DataInputStream _dis = null;
9         private DataOutputStream _dos = null;
10        private JTextArea _textarea = null;
11
12        public ClientProcessorThread(Socket socket, JTextArea text_area){
13            _socket = socket;
14            _textarea = text_area;
15            try{
16                _dos = new DataOutputStream(_socket.getOutputStream());
17                _dis = new DataInputStream(_socket.getInputStream());
18            }catch(Exception e){
19                e.printStackTrace();
20            }
21        } // end constructor
22
23        public void run(){
24            try{
25                String s = null;

```

```

26         while((s=_dis.readUTF()) != null){
27             System.out.println(s);
28             _dos.writeUTF(s);
29             _textarea.append(s + '\n');
30         }
31     }catch(Exception e){
32         e.printStackTrace();
33     }
34     finally{
35         try{
36             _socket.close();
37             _dis.close();
38             _dos.close();
39         }catch(Exception ignored){ }
40     }
41 } // end run() method
42 } // end ClientProcessorThread class definition

```

21.6 appletserver.html

```

1     <html>
2     <applet code="AppletServer.class" height=200 width=300 >
3     </html>

```

Referring to examples 21.3 through 21.6 — the AppletServer program implements a simple multi-threaded server that reads a line of UTF text, appends it to a JTextArea, and echos it back to the client program. It does this with the help of two thread classes: ServerThread and ClientProcessorThread.

The AppletServer class implements two of the four applet life-cycle methods: init() and start(). The init() method initializes the AppletServer GUI components and the start() method kicks off an instance of ServerThread on lines 18 and 19 of example 21.3.

Referring to example 21.4 — the ServerThread class is responsible for listening for incoming client connections and passing the resulting Socket object to an instance of ClientProcessorThread. The ServerThread class constructor takes a reference to a JTextArea and eventually passes the reference on to the ClientProcessorThread constructor on line 24. Attempts to connect to AppletServer from hosts other than its server will result in an AccessControlException. This exception is thrown by the ServerSocket.accept() method and handled in the catch clause on line 26.

Referring to example 21.5 — the ClientProcessorThread takes the Socket and JTextArea references as arguments to its constructor. The purpose of this class is to write client messages to the applet's JTextArea and echo them back to the client until the client disconnects or until the applet shuts down.

Example 21.6 gives the HTML code for a simple web page that hosts the AppletServer applet. Figure 21-8 shows the AppletServer applet running in a web browser and being accessed with the SimpleClient application originally presented in chapter 20.

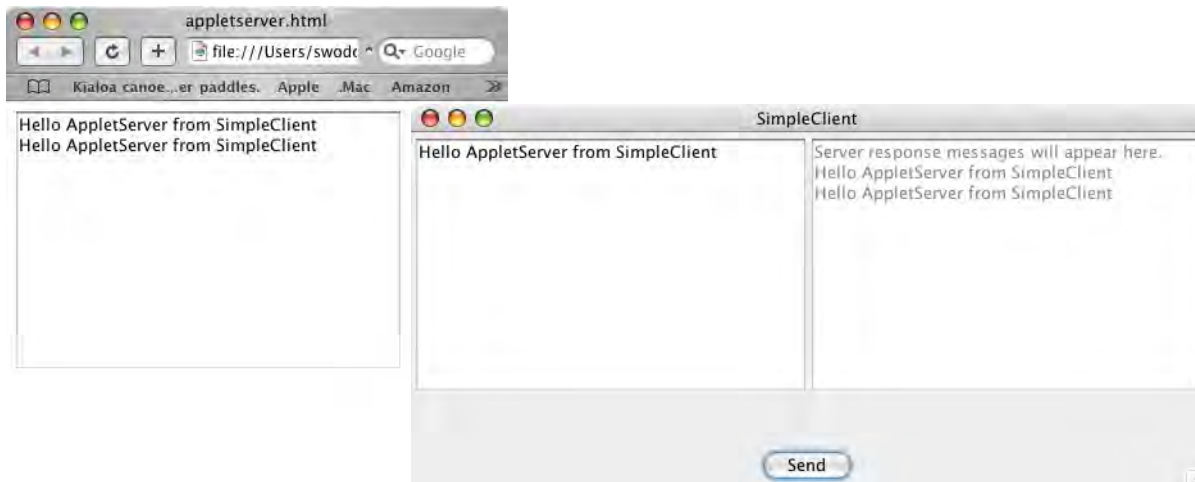


Figure 21-8: AppletServer Applet Running In A Browser And Being Accessed By The SimpleClient Application

So long as the SimpleClient application is run on the same computer from which the AppletServer applet was served it can connect and interact with the AppletServer applet as expected. However, any attempt to connect to AppletServer from a computer other than its server has the results shown in figure 21-9.

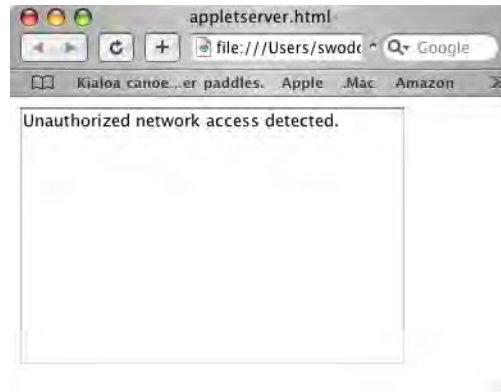


Figure 21-9: Results of Attempting to Connect to AppletServer from a Computer Other Than its Server

MISCELLANEOUS APPLET SECURITY RESTRICTIONS

In addition to file system and network access restrictions placed on untrusted applet code, applets are limited in the following ways:

- They cannot listen on network ports less than or equal to 1024
- They cannot register or create a *URLStreamHandlerFactory*, *ContentHandlerFactory*, or *SocketImplFactory*
- They cannot call `System.exit()` or `Runtime.exit()`
- They cannot call `Runtime.exec()` methods
- They cannot dynamically load library modules using the `System` or `Runtime` classes' `load()` or `loadLibrary()` methods
- Swing windows created by applets have a warning at the bottom
- They can't print, access the clipboard, or access the system event queue
- They have restricted access to certain system properties
- They cannot create or access external threads or thread groups
- They are limited in their ability to load classes
- They are limited in their ability to use reflection

SECURITY POLICIES & SIGNED APPLETS

Having presented the general case I should point out that applets execute in an environment defined by the web browser. By manipulating web browser security policies and using signed applets these security restrictions can be significantly relaxed. Further research regarding security policies and signed applets is left to you as an exercise.

QUICK REVIEW

Applets run in a restricted environment and are considered to be untrusted code. An applet cannot, as a rule, access the local file system or initiate or receive network connections to any computer other than the one from which it was served.

USING APPLET PARAMETERS

Applets can utilize parameter strings passed to it from its HTML page via the `<param>` tag. The `<param>` tag takes the following form:

```
<param name="parameter_name" value="parameter_value" >
```

The `<param>` tag has two attributes: *name* and *value*. Notice that both the *name* and *value* attributes are strings. To use a parameter in an applet you get the parameter value by calling the `Applet.getParameter(String name)` method using the parameter's name as an argument to the method call.

Examples 21.7 and 21.8 implement a short applet named `ParameterApplet` that utilizes two parameters to print a short poem by Ogden Nash in a `JTextArea`.

21.7 *ParameterApplet.java*

```

1      import javax.swing.*;
2      import java.util.*;
3
4      public class ParameterApplet extends JApplet {
5
6          private JTextArea _textarea = null;
7          private JScrollPane _scrollpane = null;
8
9          public void init(){
10             _textarea = new JTextArea(10, 20);
11             _scrollpane = new JScrollPane(_textarea);
12             this.getContentPane().add(_scrollpane);
13         }
14
15         public void start(){
16             _textarea.append(this.getParameter("WelcomeMessage") + '\n');
17             _textarea.append("\n");
18             StringTokenizer st = new StringTokenizer(this.getParameter("Poem"), "-");
19             while(st.hasMoreTokens()){
20                 _textarea.append(st.nextToken() + '\n');
21             }
22         }
23     } // end ParameterApplet class definition

```

21.8 *parameterapplet.html*

```

1      <html>
2          <applet code="ParameterApplet.class" height=300 width=300>
3              <param name="WelcomeMessage" value="A nice poem by Ogden Nash...">
4              <param name="Poem" value="Always Marry An April Girl- -Praise the spells and bless the charms, -I
found April in my arms.-April golden, April cloudy,-Gracious, cruel, tender, rowdy;-April soft in flowered
languor,-April cold with sudden anger,-Ever changing, ever true-I love April, I love you." >
5          </applet>
6      </html>

```

Referring first to example 21.7 — the `ParameterApplet` class declares a `JTextArea` and a `JScrollPane` and initializes these components in its `init()` method. On line 16 of the `start()` method it reads the first parameter named `WelcomeMessage` and displays its string value in the `JTextArea`. The next line simply appends a new line to the `JTextArea`. A `StringTokenizer` object is created on line 18 using the string value of the second parameter named `Poem`. The while loop on line 19 loops through all the `StringTokenizer`'s tokens and appends them to the `JTextArea`.

Referring to example 21.8 — the two parameters used in `ParameterApplet` are declared using the `<param>` tags on lines 3 and 4. Line 4 is one long line and is shown here wrapped. Notice how the `StringTokenizer` splits the long `Poem` parameter string into tokens based on the “-” string. Figure 21-10 shows the results of running this applet in a web browser.

Since all parameter values are strings, you may have to utilize the wrapper classes like `Integer`, `Float`, etc., to convert string values to primitive type values if you require these as parameters in your applet.

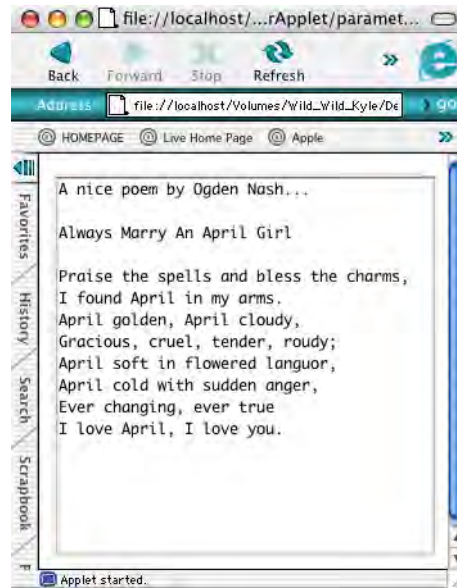


Figure 21-10: Results of Running ParameterApplet

Quick Review

Use `<param>` tags to embed applet parameters in HTML pages. Use the Applet class's `getParameter()` method to fetch parameter values and use them in your program. Use wrapper classes as required to convert parameter value strings to primitive type values.

EXTENDED APPLET EXAMPLE – POETRY IN BROWSER

This section presents an extended applet example I call Poetry In Browser. It allows you to arrange word tiles on a canvas to create short poems. The words that form the tiles are set via applet parameters. Examples 21.9 through 21.11 present the code for two classes named `Node` and `Poetry` and the HTML page that serves up the Poetry applet.

21.9 Node.java

```

1      package com.pulpfreepress.poetry;
2
3      import java.awt.*;
4      import javax.swing.*;
5      import java.lang.Math;
6
7
8      /*****
9      Node represents an individual word to be drawn on a canvas.
10     *****/
11     public class Node {
12
13         private int _oldx;
14         private int _oldy;
15         private int _x;
16         private int _y;
17         private int _h;
18         private int _w;
19         private String _its_string = null;
20         private FontMetrics _its_fontmetrics = null;
21         private final int XMARGIN = 3;
22         private final int YMARGIN = 3;
23
24
25         public Node(String the_string, Component c){
26             _its_fontmetrics = c.getFontMetrics(c.getFont());

```

```

27     _its_string = the_string;
28     _w = _its_fontmetrics.stringWidth(the_string) + (XMARGIN * 2);
29     _h = _its_fontmetrics.getHeight() + (YMARGIN * 2);
30
31     Dimension size = c.getSize();
32     _x = (int)(Math.random() * size.width);
33     _y = (int)(Math.random() * size.height);
34 }
35
36
37 public void setX(int x){
38     _oldx = _x;
39     _x = x - 10;
40 }
41
42 public void setY(int y){
43     _oldy = _y;
44     _y = y - 5;
45 }
46
47 public void paint(Graphics g){
48     g.setColor(Color.black);
49     g.fillRect(_oldx, _oldy, _w, _h);
50     g.setColor(Color.yellow);
51     g.fillRect(_x, _y, _w, _h);
52     g.setColor(Color.blue);
53     g.drawString(_its_string, _x + XMARGIN, _y + (_h/2) + YMARGIN);
54 }
55
56 public boolean inNode(int x, int y){
57     return((x > (_x)) && (x < (_x+_w + 10)) && (y > (_y)) && (y <= (_y+_h + 5)));
58 }
59
60 public String getString(){ return _its_string;}
61
62 public boolean wasTouched(int x, int y, int w, int h){
63     return( (x >= (_x - w - w*4)) && (x <= (_x+_w + w*4 )) &&
64             (y >= (_y - h - h*4)) && (y <= (_y+_h + h*4)));
65 }
66
67 public int getWidth(){return _w;}
68
69 public int getHeight(){return _h;}
70
71 }// end Node class

```

21.10 Poetry.java

```

1 package com.pulpfreepress.poetry;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import java.awt.event.*;
6 import java.util.*;
7 import com.pulpfreepress.*;
8
9 public class Poetry extends JApplet implements MouseMotionListener, MouseListener {
10
11     private Node         _nodes[];
12     private Node         _selected_node = null;
13     private int          _oldX, _oldY;
14     private Dimension    _size = null;
15     private String       _words;
16     private StringTokenizer _st;
17
18     public void init() {
19         this.setBackground(Color.black);
20         _size = this.getSize();
21         _words = new String(this.getParameter("lexicon"));
22         _st = new StringTokenizer(_words, ",");
23         int count = _st.countTokens();
24         _nodes = new Node[count];
25         for(int i=0; i<count; i++){
26             _nodes[i] = new Node(_st.nextToken().trim(), this);
27         }
28         this.addMouseMotionListener(this);
29         this.addMouseListener(this);
30         repaint();
31     } // end init() method

```

```

32
33
34     public void paint( Graphics g ) {
35         for(int i = 0; i < _nodes.length; i++) {
36             _nodes[i].paint(g);
37         }
38     }
39
40     /*****
41     MouseListener Interface Methods
42     *****/
43     public void mouseClicked(MouseEvent e){}
44     public void mouseEntered(MouseEvent e){}
45     public void mouseExited(MouseEvent e){}
46
47     public void mousePressed(MouseEvent e){
48         for(int i = 0; i < _nodes.length; i++){
49             if(_nodes[i].inNode(e.getX(), e.getY())) {
50                 _selected_node = _nodes[i];
51             } // end if
52         } // end for
53     } // end mousePressed() method
54
55     public void mouseReleased(MouseEvent e){
56         _selected_node = null;
57     }
58
59     /*****
60     MouseMotionListener Interface Methods
61     *****/
62     public void mouseDragged(MouseEvent e){
63         if(_selected_node != null){
64             _selected_node.setX(e.getX());
65             _selected_node.setY(e.getY());
66             _selected_node.paint(this.getGraphics());
67             for(int i = 0; i < _nodes.length; i++) {
68                 if(_nodes[i].wasTouched(e.getX(), e.getY(), _selected_node.getWidth(),
69                     _selected_node.getHeight())) {
70                     _nodes[i].paint(this.getGraphics());
71                 } // end if
72             } // end for
73         } // end if
74     } // end mouseDragged() method
75
76     public void mouseMoved(MouseEvent e){}
77
78 } // end Poetry class definition

```

21.11 PoetryApplet.html

```

79 <title>Poetry In Browser</title>
80 <hr>
81 <applet archive="PoetryClasses.jar" code="com.pulpfreepress.poetry.Poetry.class" width=600 height=600>
82 <PARAM name="lexicon" value="hate,love,love,passion,desire,sex,you,
83     us,we,we,I,I,need,you,to,love,me,forever,tonight,the,our,
84     passion,will,rage,s,'s,sheets,flannel,wrap,wrapped,fan,flames,
85     fire,heat,night,sun,moon,stars,shine,bright,and,cast,an,image,
86     upon,my,soul,touch,reach,beyond,and,but,or,class,back,seat,butt,
87     tight,wet,make,to,run,swim,ing,less,less,regret,shack,up,down,go,
88     round,round,tonight,make,love,fill,full,ing,">
89 </applet>
90 <hr>
91 <a href="Poetry.java">The source</a>

```

Referring to example 21.9 — the Node class represents an individual word to be displayed in the Poetry applet. Example 21-10, the Poetry applet, reads the lexicon parameter from the HTML page and creates an array of Nodes, which are painted onto the applet. The words that make up the lexicon can be easily changed by changing the lexicon parameter's value string on the HTML page.

Referring to example 21.11 — the Poetry and Node classes are packaged in a jar file named PoetryClasses.jar. The fully qualified name of the Poetry class is “com.pulpfreepress.poetry.Poetry.class”. Note that the <html> and </html> tags are optional and are not part of the HTML code in this example.

Figure 21-11 shows the Poetry applet running in a web browser with a short verse arranged.

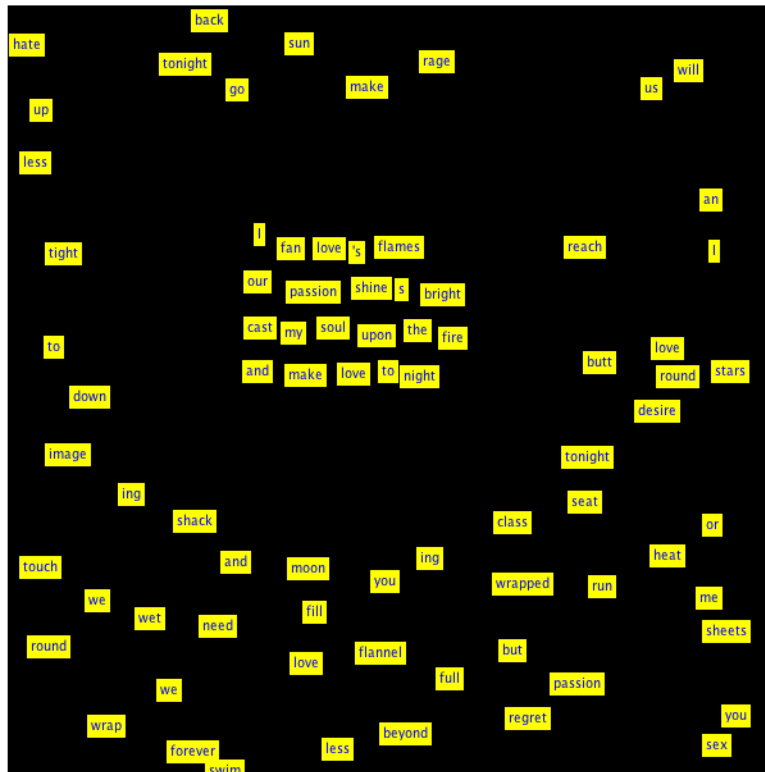


Figure 21-11: Poetry Applet In Action

JAVA DATABASE CONNECTIVITY (JDBC™) OVERVIEW

The JDBC API is a collection of Java classes and interfaces that enables you to access, manipulate, and persist data to a data source. Most of the time the data source is a relational database like Oracle, Informix, or MySQL. (*There are many others!*) However, the data source could very well be a spreadsheet, a flat file, or some other type of tabular data.

JDBC™ PACKAGES

The JDBC API comes in two packages: *java.sql* and *javax.sql*. The *java.sql* package, referred to as the *JDBC core API*, provides the bulk of the JDBC functionality while the *javax.sql* package, referred to as the *JDBC Optional Package API*, provides server-side JDBC support. The JDBC programming examples in this chapter will utilize a subset of the interfaces and classes found in the *java.sql* package.

JDBC™ SPECIFICATION

JDBC is also a specification. Most of what the API provides is in the form of interfaces. Java platform vendors like Sun or Apple must provide implementation classes to fulfill the contract specified by these interfaces. Database vendors are also responsible for providing suitable JDBC driver classes that allow Java programs to connect to their databases via JDBC.

Using JDBC – A Macro View

Using JDBC to connect and interact with a database is like solving a puzzle that has the following pieces:

- A database that's been properly configured (*i.e.*, tables, relationships, access rights, etc.)
- A suitable database JDBC driver class (*provided by the database vendor*)
- Knowledge of relational database design (*if you are accessing a relational database*)
- Knowledge of the database schema you are going to access
- Knowledge of SQL
- A Java program that uses the JDBC API to connect to the database

Properly Configured Database

If you own the database then you will have to set things up yourself. This means you must get smart on how to install and configure your particular vendor's database product.

WARNING: *Not all things potentially possible with JDBC are supported by all database products.* For instance, JDBC provides a way to execute stored procedures. If your database product does not support stored procedures then you will not be able to use this aspect of JDBC.

JDBC Driver Class

The database vendor will supply the JDBC driver. However, you must track down the driver class(es) or jar file and make sure it is installed on your machine and that the classpath is properly set to find it. If you can't find this information in your database's documentation chances are you can find it easily on the internet.

Knowledge of Relational Database Design

If you are planning to use JDBC to access a relational database then you must possess, at the very least, a rudimentary understanding of relational database concepts. You must understand the concepts of tables, columns, and rows. You must understand the concepts of primary keys and foreign keys and how they are used to establish relationships between tables.

Knowledge of the Database Schema

As I said above, there's no need to be an expert in relational database design theory — you just need to know a little something about it so you can read and understand the database schema against which you will write your JDBC code.

Knowledge of SQL

Your JDBC code will include Structured Query Language (SQL) statements. You must know the difference between Data Definition Language (DDL) statements and Data Manipulation Language (DML) statements. You must get comfortable writing select, insert, update, and delete statements.

The Java JDBC Program

Lastly, all this knowledge comes together in the form of a Java program that utilizes the JDBC API to connect to and manipulate data in a relational database. If your program is relatively simple, like the primary JDBC program presented in this chapter, then you will survive well by arming yourself with a basic understanding of all these subject areas. However, increasingly complex projects demand a corresponding increase in competence in each area.

GENERAL STEPS REQUIRED TO EMPLOY JDBC

JDBC is used in a Java program to establish a connection to a database for the purposes of accessing, manipulating, or storing data. The following general steps describe the process of using JDBC to connect to a database and execute SQL statements:

Step 1: Load the database driver using the `Class.forName()` method

Step 2: Use the `DriverManager` class's `getConnection()` method to create a `Connection` object

Step 3: Use the `Connection` object to create a `Statement` or `PreparedStatement` object as required

Step 4: Use the `Statement` or `PreparedStatement` object to submit an SQL statement to the database

Step 5: If applicable, use the `ResultSet` object obtained from step 4 to process the retrieved data

Each of these steps can be executed at different locations within your program at different times. For instance, you usually have to load the JDBC driver class just once. In complex applications this may be already done for you as part of the application's environment initialization. You generally only need to establish the connection to the database once as well and keep it open as long as you need access to the database. Statements, including `PreparedStatement`s, might need to be created and closed many times during the course of application execution.

MOVING FORWARD

As you have by now concluded, learning to use JDBC dictates that you must learn a few new tricks as well. If you are a complete novice in the areas of relational database concepts and SQL then I recommend you procure a few reference books on those topics and study up. I have listed several good ones at the end of this chapter. I will attempt to explain, as simply as I can, what you need to know about these topics, so you can move forward with the remaining chapter material.

Quick Review

JDBC™ is an API that lets you access data sources from Java programs. The JDBC API comes in two packages: the `java.sql` package contains the JDBC core API and the `javax.sql` package contains the JDBC Optional Package API. The JDBC API consists mostly of interfaces which must be implemented by Java platform vendors. Database vendors must also supply JDBC driver classes to facilitate JDBC connectivity to their database products.

JDBC PROJECT DESCRIPTION

The remaining sections in this chapter will present and discuss JDBC concepts in the context of an extended programming example that implements an employee training management system.

Overall System Architecture

The architecture diagram for this system is shown in figure 21-12.

Referring to figure 21-12 — an applet will execute in a client-side browser and communicate via the internet using RMI with a server application running on a remote host. The server application will utilize JDBC to connect to a MySQL database running on the same machine.

Recall, if you will, the discussion from chapter 19 concerning the logical and physical distribution of networked applications. The architecture presented above could easily be arranged so the database resides on its own computer or, for testing purposes, all three components could execute on the same computer as well.

Detailed System Architecture

Figure 21-13 shows a detailed class diagram for the server-side components.

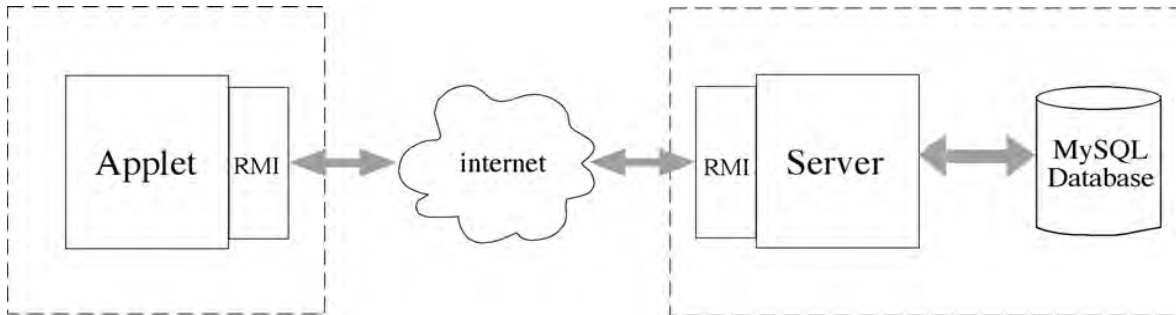


Figure 21-12: Employee Training Management System Architecture Diagram

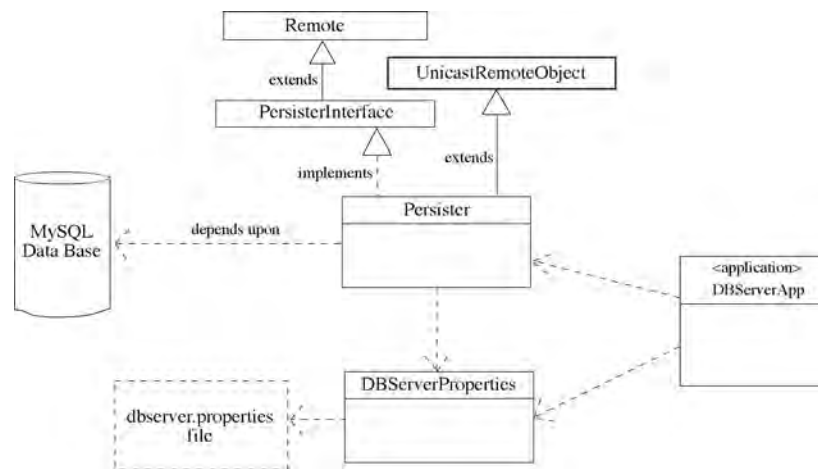


Figure 21-13: Server-Side Component Class Diagram

Referring to figure 21-13 — the DBServerApp class is the main application class for the server-side component of the employee training management system. The DBServerApp class depends upon the Persist and the DBServer-Properties classes.

The Persist class is an RMI component that uses JDBC to connect to an instance of a MySQL database. The Persist class will translate RMI method calls it receives from the applet into JDBC calls to the employee training database. The Persist class also depends upon the DBServerProperties class.

Figure 21-14 shows the class diagram for the client-side applet component.

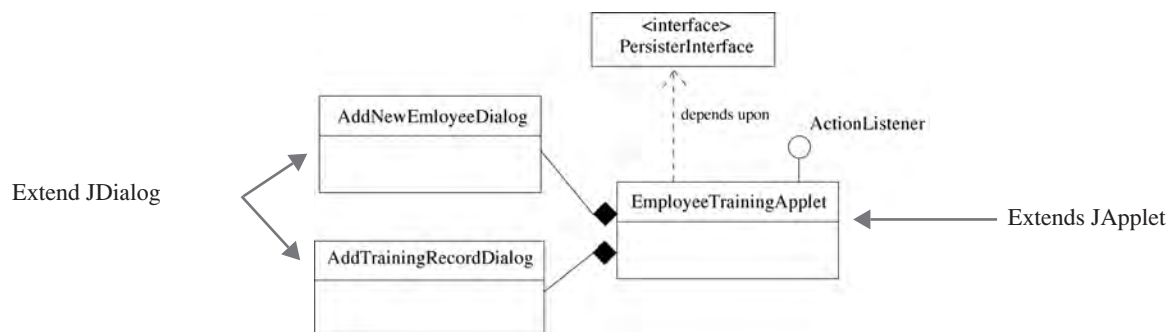


Figure 21-14: Client-Side Component Class Diagram

Referring to figure 21-14 — the EmployeeTrainingApplet class depends upon the PersistInterface. It contains two dialogs: AddNewEmployeeDialog and AddTrainingRecordDialog. What is not shown in the diagram is the inheritance relationship between EmployeeTrainingApplet and JApplet, and the two dialogs and JDialog.

ROLE OF THE PERSISTER CLASS

The Persister class is the key component on the server-side. It implements PersisterInterface which is given in example 21.12 below.

21.12 PersisterInterface.java

```
1     package com.pulpfreepress.dobserver;
2
3     import java.rmi.*;
4     import java.util.*;
5     import com.pulpfreepress.business_objects.*;
6
7     public interface PersisterInterface extends Remote {
8         public Vector queryByLastName(String last_name) throws RemoteException;
9         public void addNewEmployee(Employee emp) throws RemoteException;
10        public void addTrainingRecord(EmployeeTraining et) throws RemoteException;
11        public Employee getEmployeeTrainingRecords(Employee emp) throws RemoteException;
12    }
```

Referring to example 21.12 — PersisterInterface specifies four methods: queryByLastName(), addNewEmployee(), addTrainingRecord(), and getEmployeeTrainingRecords(). The types Employee and EmployeeTraining are classes that were created to represent Employee and EmployeeTraining objects respectively. These classes, along with the project's package structure, will be discussed in detail below. For now I just want you to be aware that these four methods specified by PersisterInterface represent the total sum of functionality of the employee training management system. That is, new employee records can be created, the database can be searched for employees by last name, new employee training records can be added, and all of an employee's training records can be retrieved. It is the role of the Persister class to deliver this functionality.

Quick Review

In this section I presented an overall architectural view of the employee training management system that will be implemented using Java applet and JDBC technology. It is time now to discuss MySQL, relational database, SQL, and JDBC concepts in more detail to set the stage for your understanding of the project and its code.

MySQL DATABASE & JDBC

The employee training management system persists its data in a MySQL relational database with the help of JDBC. In this section I want to focus your attention on those aspects of MySQL, relational database design, SQL, and JDBC that are critical to your understanding of the employee training management system and its supporting code.

SETTING UP MySQL TO RUN THE EXAMPLE CODE

Proper installation and configuration of the MySQL database is the primary piece of the puzzle and requires executing the following general steps:

- Step 1:** Obtain and install the product
- Step 2:** Create a database named chapter_21
- Step 3:** Establish the required access permissions to the chapter_21 database
- Step 4:** Create two tables within the chapter_21 database named employees and employee_training
- Step 5:** Establish the required access permissions to the tables

These steps are discussed in greater detail below.

OBTAINING AND INSTALLING MySQL

If you don't currently have MySQL installed on your computer you will need to do so before proceeding with database and table creation. MySQL is available for a variety of platforms and can be obtained from the MySQL website. [<http://www.mysql.com/>] The MySQL installation instructions are excellent and I refer you to them to perform the actual installation. I strongly recommend you have a piece of paper and pen ready to take notes regarding installation location and other important installation information.

When installation is complete you should be able to execute two programs from the command line: *mysqldadmin* and *mysql*.

THE *mysqldadmin* PROGRAM

To test the installation open a terminal or command prompt window and type *mysqldadmin*. You should see something that resembles the following output:

21.13 *mysqldadmin help*

```

1      [Rick-Millers-Computer:-] swodog% mysqldadmin
2      mysqldadmin Ver 8.4.1 Distrib 4.1.10, for apple-darwin7.7.0 on powerpc
3      Copyright (C) 2000 MySQL AB & MySQL Finland AB & TCX DataKonsult AB
4      This software comes with ABSOLUTELY NO WARRANTY. This is free software,
5      and you are welcome to modify and redistribute it under the GPL license
6
7      Administration program for the mysqld daemon.
8      Usage: mysqldadmin [OPTIONS] command command...
9      -c, --count=#          Number of iterations to make. This works with -i
10     (--sleep) only.
11     -#, --debug[=name]    Output debug log. Often this is 'd:t:o,filename'.
12     -f, --force           Don't ask for confirmation on drop database; with
13     multiple commands, continue even if an error occurs.
14     -C, --compress       Use compression in server/client protocol.
15     --character-sets-dir=name
16     Directory where character sets are.
17     --default-character-set=name
18     Set the default character set.
19     -?, --help           Display this help and exit.
20     -h, --host=name      Connect to host.
21     -p, --password[=name]
22     Password to use when connecting to server. If password is
23     not given it's asked from the tty.
24     -P, --port=#        Port number to use for connection.
25     --protocol=name     The protocol of connection (tcp,socket,pipe,memory).
26     -r, --relative      Show difference between current and previous values when
27     used with -i. Currently works only with extended-status.
28     -O, --set-variable=name
29     Change the value of a variable. Please note that this
30     option is deprecated; you can set variables directly with
31     --variable-name=value.
32     -s, --silent        Silently exit if one can't connect to server.
33     -S, --socket=name   Socket file to use for connection.
34     -i, --sleep=#      Execute commands again and again with a sleep between.
35     -u, --user=name     User for login if not current user.
36     -v, --verbose      Write more information.
37     -V, --version      Output version information and exit.
38     -E, --vertical     Print output vertically. Is similar to --relative, but
39     prints output vertically.
40     -w, --wait[=#]     Wait and retry if connection is down.
41     --connect_timeout=#
42     --shutdown_timeout=#
43
44     Variables (--variable-name=value)
45     and boolean options {FALSE|TRUE}  Value (after reading options)
46     -----
47     count                          0
48     force                          FALSE
49     compress                        FALSE
50     character-sets-dir              (No default value)
51     default-character-set           (No default value)
52     host                            (No default value)
53     port                            3306
54     relative                        FALSE
55     socket                          (No default value)
56     sleep                           0
57     user                            (No default value)
58     verbose                        FALSE
59     vertical                        FALSE

```

```

60     connect_timeout      43200
61     shutdown_timeout    3600
62
63     Default options are read from the following files in the given order:
64     /etc/my.cnf /usr/local/mysql/data/my.cnf ~/.my.cnf
65     The following groups are read: mysqladmin client
66     The following options may be given as the first argument:
67     --print-defaults      Print the program argument list and exit
68     --no-defaults        Don't read default options from any options file
69     --defaults-file=#     Only read default options from the given file #
70     --defaults-extra-file=# Read this file after the global files are read
71
72     Where command is a one or more of: (Commands may be shortened)
73     create databasename  Create a new database
74     debug                Instruct server to write debug information to log
75     drop databasename    Delete a database and all its tables
76     extended-status      Gives an extended status message from the server
77     flush-hosts          Flush all cached hosts
78     flush-logs           Flush all logs
79     flush-status         Clear status variables
80     flush-tables         Flush all tables
81     flush-threads        Flush the thread cache
82     flush-privileges     Reload grant tables (same as reload)
83     kill id,id,...       Kill mysql threads
84     password new-password Change old password to new-password, MySQL 4.1 hashing.
85     old-password new-password Change old password to new-password in old format.
86
87     ping                 Check if mysqld is alive
88     processlist          Show list of active threads in server
89     reload               Reload grant tables
90     refresh              Flush all tables and close and open logfiles
91     shutdown            Take server down
92     status               Gives a short status message from the server
93     start-slave          Start slave
94     stop-slave           Stop slave
95     variables            Prints variables available
96     version              Get version info from server

```

Referring to example 21.13 — this listing represents the help output from executing the `mysqladmin` program with no command-line input. The `mysqladmin` program is used to set database operational parameters and to execute routine database maintenance commands. An important piece of information to take note of is the port number the database is listening on. This information is located on line 53 and is set to port number 3306.

If all these different settings and commands seem confusing at first don't worry. The default values are fine set just the way they are for the purposes of this chapter.

THE *mysql* MONITOR PROGRAM

The `mysql` monitor program is the interactive SQL interpreter through which you enter SQL commands to interact with MySQL databases. When MySQL is first installed it is pre-configured with two databases: *mysql* and *test*. To verify this type `mysql` at the command line and hit return. You should see something similar to figure 21-15.

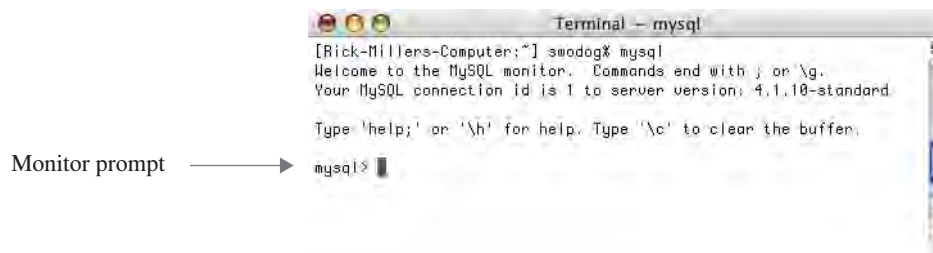
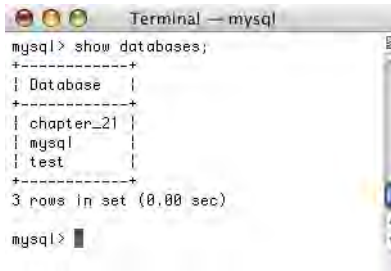


Figure 21-15: MySQL Monitor Program on Startup

You can enter SQL and MySQL commands at the `mysql` monitor prompt. To verify the installed databases type “*show databases*” at the monitor prompt followed by a semicolon and then hit return. The output you see should look similar to figure 21-16.

Referring to figure 21-16 — three databases are listed: *chapter_21*, which was created to support the employee training management system (you will not see it listed until after you create it), *mysql*, and *test*. The `mysql` database stores admin tables used by the MySQL database application.



```

mysql> show databases;
+-----+
| Database |
+-----+
| chapter_21 |
| mysql |
| test |
+-----+
3 rows in set (0.00 sec)

mysql>

```

Figure 21-16: Results of Entering “show databases;” at the Monitor Prompt



```

mysql> use mysql;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -R

Database changed
mysql> show tables;
+-----+
| Tables_in_mysql |
+-----+
| columns_priv |
| db |
| func |
| help_category |
| help_keyword |
| help_relation |
| help_topic |
| host |
| tables_priv |
| time_zone |
| time_zone_leap_second |
| time_zone_name |
| time_zone_transition |
| time_zone_transition_type |
| user |
+-----+
15 rows in set (0.00 sec)

mysql>

```

Figure 21-17: Results of Changing to the mysql Database with “use mysql;” and Entering “show tables;”

To use a particular database enter “*use*” followed by the database name and a semicolon at the monitor prompt then hit enter. Next, enter “*show tables*” followed by a semicolon at the monitor prompt then hit enter. You should see an output similar to figure 21-17

Helpful MySQL Monitor Commands

Table 21-1 lists and describes several mysql monitor commands you will find helpful. (*Note: All commands except exit and quit are followed by a semicolon.*)

Command	Description
show databases	Lists the databases available to the MySQL monitor.
use <i>database_name</i>	Directs SQL commands against named database.
show tables	Lists the tables contained in the current database.
describe <i>table_name</i>	Lists table column attributes for named table.
exit	Exit MySQL monitor program.
quit	Exit MySQL monitor program.

Table 21-1: Helpful MySQL Monitor Commands

CREATING THE CHAPTER_21 DATABASE

Before you can use the `chapter_21` database it must be created using the `mysqladmin` program. Refer to the `mysqladmin` help listed in example 21.13 and find the `create databasename` command located on line 73. To create the `chapter_21` database type “`mysqladmin create chapter_21`” at a terminal or command prompt and hit return. If all goes well the terminal or command prompt will return within a second or two as if nothing happened. To check that the `chapter_21` database was indeed created start the `mysql` monitor program and enter the `show databases` command. Your listing should now look like figure 21-16 and list three databases: `chapter_21`, `mysql`, and `test`.

Establishing Database Security Via MySQL Access Control Tables

Now that the `chapter_21` database is created you must establish usage permissions. To do this you must be aware of five tables located within the `mysql` database: `user`, `db`, `tables_priv`, `columns_priv`, and `host`. To control access to a database you will make an entry into one or more of these tables. To switch to the `mysql` database enter “`use mysql;`” at the `mysql` program monitor prompt.

In a nutshell here’s what you need to do. You need to add a user to the `user` table who has less privileges than root, then create the required tables in the `chapter_21` database, then set `db` permissions, optionally set table permissions, and finally, optionally set column permissions if required.

Adding A New User To The User Table

If you issue “`describe user;`” at the monitor prompt you will see the complete structure of the `user` table. Figure 21-18 shows the structure of the `user` table.

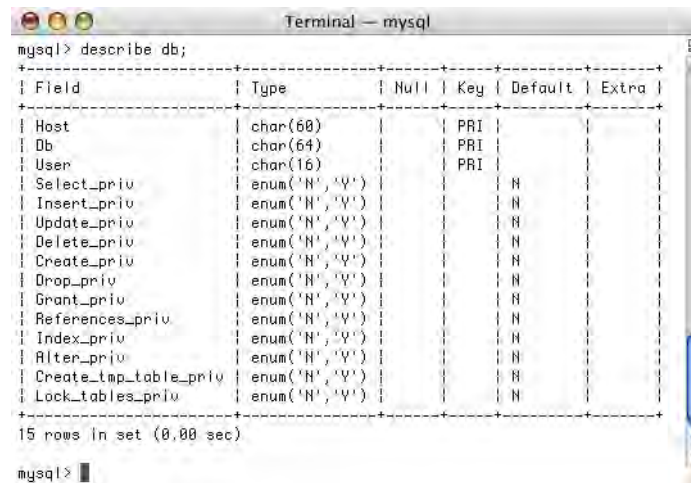
```
mysql> describe user;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| Host | varchar(60) | | PRI | | |
| User | varchar(16) | | PRI | | |
| Password | varchar(41) | | | | |
| Select_priv | enum('N','Y') | | | N | |
| Insert_priv | enum('N','Y') | | | N | |
| Update_priv | enum('N','Y') | | | N | |
| Delete_priv | enum('N','Y') | | | N | |
| Create_priv | enum('N','Y') | | | N | |
| Drop_priv | enum('N','Y') | | | N | |
| Reload_priv | enum('N','Y') | | | N | |
| Shutdown_priv | enum('N','Y') | | | N | |
| Process_priv | enum('N','Y') | | | N | |
| File_priv | enum('N','Y') | | | N | |
| Grant_priv | enum('N','Y') | | | N | |
| References_priv | enum('N','Y') | | | N | |
| Index_priv | enum('N','Y') | | | N | |
| Alter_priv | enum('N','Y') | | | N | |
| Show_db_priv | enum('N','Y') | | | N | |
| Super_priv | enum('N','Y') | | | N | |
| Create_tmp_table_priv | enum('N','Y') | | | N | |
| Lock_tables_priv | enum('N','Y') | | | N | |
| Execute_priv | enum('N','Y') | | | N | |
| Repl_slave_priv | enum('N','Y') | | | N | |
| Repl_client_priv | enum('N','Y') | | | N | |
| ssl_type | enum('', 'ANY', 'X509', 'SPECIFIED') | | | | |
| ssl_cipher | blob | | | | |
| x509_issuer | blob | | | | |
| x509_subject | blob | | | | |
| max_questions | int(11) unsigned | | | 0 | |
| max_updates | int(11) unsigned | | | 0 | |
| max_connections | int(11) unsigned | | | 0 | |
+-----+-----+-----+-----+-----+-----+
31 rows in set (0.00 sec)
```

Figure 21-18: Structure of the `user` Table Located in the `mysql` Database

Referring to figure 21-18 — the structure of the `user` table is fairly straightforward. The `Host`, `User`, and `Password` columns each take a string. The various privilege columns (*i.e.*, `Select_priv`, `Insert_priv`, etc.) each take either the character ‘Y’ representing *yes* or the character ‘N’ representing *no*. I recommend that you create a new user that has fewer privileges than root for security’s sake but enough privileges to perform selects, inserts, updates, and deletes. To insert a new user into the `user` table you use the SQL `insert` command.

CREATING DB PERMISSIONS

After you create a new database you'll need to set permissions on that database by adding an entry into the db table. Figure 21-19 shows the structure of the db table.



```
mysql> describe db;
```

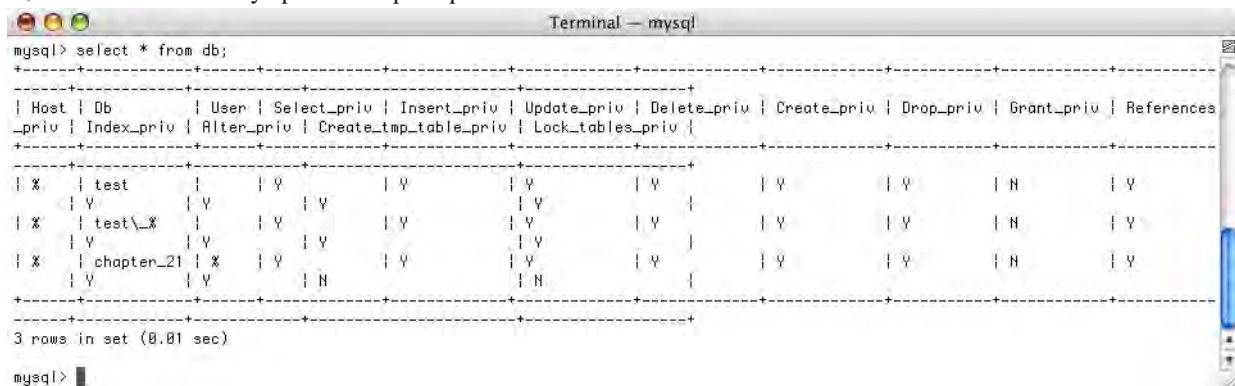
Field	Type	Null	Key	Default	Extra
Host	char(60)		PRI		
Db	char(64)		PRI		
User	char(16)		PRI		
Select_priv	enum('N','Y')			N	
Insert_priv	enum('N','Y')			N	
Update_priv	enum('N','Y')			N	
Delete_priv	enum('N','Y')			N	
Create_priv	enum('N','Y')			N	
Drop_priv	enum('N','Y')			N	
Grant_priv	enum('N','Y')			N	
References_priv	enum('N','Y')			N	
Index_priv	enum('N','Y')			N	
Alter_priv	enum('N','Y')			N	
Create_tmp_table_priv	enum('N','Y')			N	
Lock_tables_priv	enum('N','Y')			N	

```
15 rows in set (0.00 sec)

mysql>
```

Figure 21-19: Structure of the db Table

Referring to figure 21-19 — for a particular database you can specify an authorized host and user along with the required permissions. To authorize access to a database from any host by any user set the Host and User values to the wildcard character '%'. Figure 21-20 shows the contents of the db table listed as a result of issuing the “*select * from db;*” command at the mysql monitor prompt.



```
mysql> select * from db;
```

Host	Db	User	Select_priv	Insert_priv	Update_priv	Delete_priv	Create_priv	Drop_priv	Grant_priv	References_priv	Index_priv	Alter_priv	Create_tmp_table_priv	Lock_tables_priv
%	test	%	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y
%	test_%	%	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y
%	chapter_21	%	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y

```
3 rows in set (0.01 sec)

mysql>
```

Figure 21-20: Contents of the db Table

Referring to figure 21-20 — the output shown here is a somewhat hard to read because the rows have wrapped due to the length of each row, however, you should be able to correlate the table structure shown in figure 21-19 with what you see here. The db table shows permissions for two databases: test and chapter_21. The '%' in the Host column indicates that any host has access to the database. A '%' in the User column indicates that any user has access as well. The other privilege columns are set as required. For the chapter_21 database I have allowed access from any host by any user with almost all privileges set to 'Y'.

A blank entry in the Host or User column forces MySQL to verify that access has been granted to the host or user in question by checking the user and host tables for the required entry. The host table is discussed below.

CREATING TABLE PERMISSIONS

After you've created a table within a database you can optionally set permissions on that table by making an entry in the tables_priv table. Figure 21-21 shows the structure of the tables_priv table. Referring to figure 21-21 — you establish table privileges by host, database, user, table name, and granter. Granter must be an object within a MySQL database that has grant privileges. Again, use the SQL *insert* command to insert entries into the tables_priv table.

```
mysql> describe tables_priv
-> ;
```

Field	Type	Null	Key	Default	Extra
Host	char(60)		PRI		
Db	char(64)		PRI		
User	char(16)		PRI		
Table_name	char(64)		PRI		
Grantor	char(77)		PRI		
Timestamp	timestamp	YES	NUL	CURRENT_TIMESTAMP	
Table_priv	set('Select','Insert','Update','Delete','Create','Drop','Grant','References','Index','Alter')				
Column_priv	set('Select','Insert','Update','References')				

Figure 21-21: Structure of the tables_priv Table

CREATING COLUMN PERMISSIONS

You can also optionally set privileges at the granularity of a column. Figure 21-22 shows the structure of the columns_priv table.

```
mysql> describe columns_priv
-> ;
```

Field	Type	Null	Key	Default	Extra
Host	char(60)		PRI		
Db	char(64)		PRI		
User	char(16)		PRI		
Table_name	char(64)		PRI		
Column_name	char(64)		PRI		
Timestamp	timestamp	YES		CURRENT_TIMESTAMP	
Column_priv	set('Select','Insert','Update','References')				

7 rows in set (0.00 sec)

Figure 21-22: Structure of the columns_priv Table

Referring to figure 21-22 — you can grant column privileges based on host, database, and user.

GRANTING ACCESS ON A PER HOST BASIS

An additional permissions table named Host allows you to grant access to a database on a host-by-host basis. Figure 21-23 shows the structure of the Host table.

```
mysql> describe host;
-> ;
```

Field	Type	Null	Key	Default	Extra
Host	char(60)		PRI		
Db	char(64)		PRI		
Select_priv	enum('N','Y')			N	
Insert_priv	enum('N','Y')			N	
Update_priv	enum('N','Y')			N	
Delete_priv	enum('N','Y')			N	
Create_priv	enum('N','Y')			N	
Drop_priv	enum('N','Y')			N	
Grant_priv	enum('N','Y')			N	
References_priv	enum('N','Y')			N	
Index_priv	enum('N','Y')			N	
Alter_priv	enum('N','Y')			N	
Create_tmp_table_priv	enum('N','Y')			N	
Lock_tables_priv	enum('N','Y')			N	

14 rows in set (0.05 sec)

Figure 21-23: Structure of the host Table

GENERAL STRATEGY FOR MANAGING PERMISSIONS USING THE MySQL ACCESS CONTROL TABLES

All this talk of access control tables, permissions, and privileges may seem confusing at first but don't despair. You don't need to use every table to get simple examples working. For personal use in a learning environment you can pretty much throw access control out the window. If you're the only one accessing the database give yourself

sweeping access rights in the user and db tables and be done with it. If, on the other hand, you want to establish fine grain access control down to the column level you can apply the following general strategy:

- Establish user accounts with all privileges set to 'N'. This grants connect access to the MySQL server but allows no other actions
- If required, establish host access rights in the host table
- Establish access to individual databases on a per-user or per-host basis, setting privileges as required (*Use the '%' and blank values as required for finer control*)
- Establish access permissions on a per-table basis using the tables_priv table
- Establish access permissions on a per-column basis using the columns_priv table

CREATING THE EMPLOYEE TRAINING MANAGEMENT SYSTEM TABLES

Now that you have a general understanding of how to establish and control access permissions to a MySQL database it's time to create the tables that will be used in the employee training management system. Figure 21-24 shows the entity diagram expressing the relationship between an Employee and an Employee_Training record.

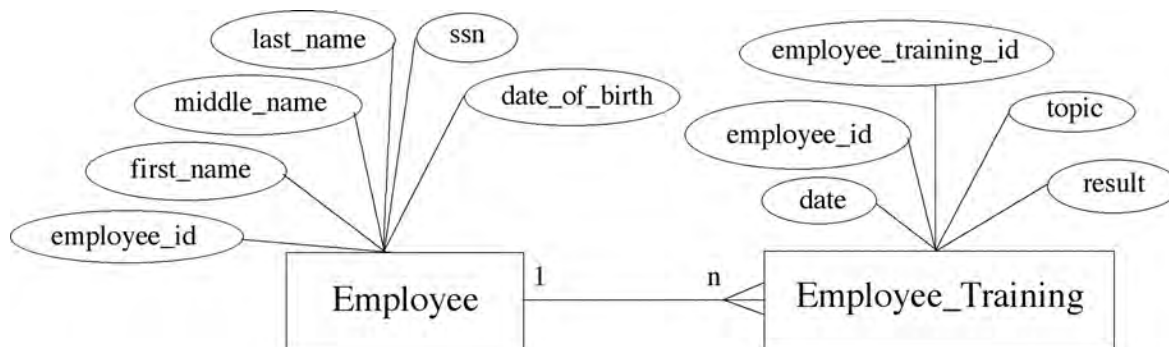


Figure 21-24: Entity Diagram for Employee and Employee_Training Tables

Referring to figure 21-24 — an employee is described by a set of attributes including *employee_id*, which is the primary key, *first_name*, *middle_name*, *last_name*, *ssn*, and *date_of_birth*. An *employee_training* record contains the attributes *employee_training_id* (primary key), *employee_id* (foreign key), *date*, *topic*, and *result*. A one-to-many relationship is established between an employee and *employee_training* via the *employee_id* foreign key attribute in the *employee_training* entity.

These entities map to tables. These tables, with their requisite columns and relationships, can be created in the `chapter_21` database by issuing SQL commands at the `mysql` monitor prompt. They can also be created using a script file like the one shown in example 21.14.

21.14 *setup_tables.sql*

```

1      # run this script to create the tables required to support the employee
2      # training tracking system example described in chapter 21 of
3      # Java For Artists.
4      create table employees
5      (
6          EMPLOYEE_ID    INT    NOT NULL PRIMARY KEY AUTO_INCREMENT,
7          FIRST_NAME     VARCHAR(50),
8          MIDDLE_NAME    VARCHAR(50),
9          LAST_NAME      VARCHAR(50),
10         SSN             VARCHAR(11),
11         DATE_OF_BIRTH  DATE,
12         Unique (SSN)
13     );
14
15     create table employee_training
16     (
17         EMPLOYEE_TRAINING_ID INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
18         EMPLOYEE_ID        INT NOT NULL REFERENCES employees (EMPLOYEE_ID),
19         DATE                DATE,
20         TOPIC               VARCHAR(200),
21         RESULT              VARCHAR(20)
22     );
23
  
```

Referring to example 21.14 — an SQL script can contain any number of comments and commands. Lines 1 through 3 contain comments. The employees table is created with the create table command starting on line 4 and ending on line 13. Notice that the command is terminated on line 13 with a semicolon. The employees table's columns are defined on lines 6 through 11. Line 12 applies the Unique() function to the SSN column. This will ensure that an employee's ssn is unique from all the other existing ssns when entered. Several column modifiers are applied to the EMPLOYEE_ID column to flag it as the primary key, ensure it is not null, and have it automatically increment in value with each new entry.

The employee_training table is created in similar fashion beginning on line 15. The relationship between this table and the employees table is established on line 18 with the use of the references modifier.

EXECUTING THE SETUP_TABLES.SQL SCRIPT

The setup_tables script can be executed by entering the following command at a terminal or command prompt:

```
mysql chapter_21 < setup_tables.sql
```

If you have problems running the script you can check a couple of things: 1) make sure the script is located in the current working directory or give the fully qualified path name to the script when you run the command, 2) make sure you have sufficient permissions to create tables in the chapter_21 database. You will get an error message indicating otherwise if you do not. If you get the permissions error after you've made what you believe are correct entries into the user and db tables try restarting MySQL.

After you run the script you can check the existence of the tables by starting the mysql monitor program, switching to the chapter_21 database, and issuing the "show tables;" command. When you do this you should see something similar to the listing shown in figure 21-25.



Figure 21-25: Newly Created chapter_21 Database Tables

POPULATING TABLES WITH DATA AND RUNNING QUERIES

To further test the chapter_21 database tables let's populate them with sample data and run some queries. To do this you'll be using the SQL *insert* command and various forms of the *select* statement. These SQL statements can be entered via the mysql monitor program.

INSERTING DATA INTO TABLES

To insert data into the employees table you can issue the SQL *insert* command. Study the following example:

21.15 insert command

```

1      insert into employees (employee_id, first_name, middle_name, last_name, ssn, date_of_birth)
2      values (0, "Homer", "J", "Simpson", "123-34-2345", "1980-09-21");

```

Referring to example 21-15 — the SQL insert command is used to insert a row of values into a database table. The list of columns affected is given between the parentheses on line 1. The values to insert into each column are given between the parentheses on line 2. The employee_id value is set to 0 because the employee_id value will automatically increment when the insert is executed.

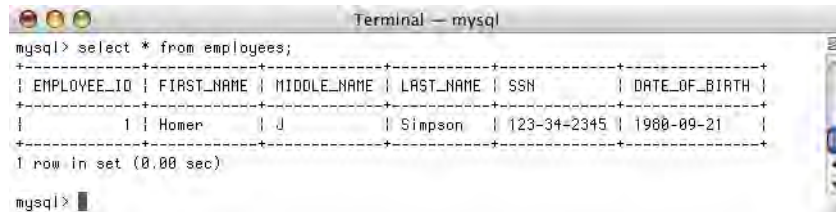
QUERYING A TABLE WITH THE SELECT STATEMENT

The contents of a database table can be examined (*queried*) with an SQL select statement. The select statement has three clauses: 1) *select* - to indicate what columns to retrieve, 2) *from* - to indicate what tables to retrieve the columns from, and 3) *where* - to specify a qualification criteria.

To select all the columns and all the rows from the employees tables you can issue the following select statement:

```
select * from employees;
```

In this example the where clause is omitted. Figure 21-26 shows the results of running this command against the employees tables in its current state.



```
Terminal - mysql
mysql> select * from employees;
+-----+-----+-----+-----+-----+-----+
| EMPLOYEE_ID | FIRST_NAME | MIDDLE_NAME | LAST_NAME | SSN      | DATE_OF_BIRTH |
+-----+-----+-----+-----+-----+-----+
| 1           | Homer     | J           | Simpson  | 123-34-2345 | 1980-09-21    |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

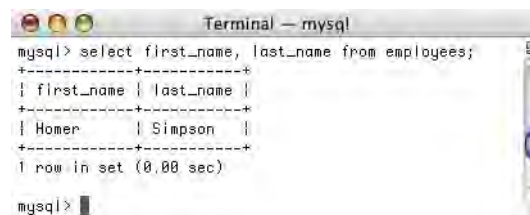
mysql>
```

Figure 21-26: Results of Executing the Select Statement Against the employees Table

Referring to example 21-26 — the *select * from employees;* statement returns all the columns and all the rows of data contained in the employees table, which, at this point, only contains one row. If you only wanted to specify certain columns you can do so in the select clause. The following select statement returns the first_name and last_name columns for all the rows in the employees table:

```
select first_name, last_name
from employees;
```

Figure 21-27 shows the results of executing this select statement.



```
Terminal - mysql
mysql> select first_name, last_name from employees;
+-----+-----+
| first_name | last_name |
+-----+-----+
| Homer     | Simpson  |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Figure 21-27: Results of Selecting Only the First_Name and Last_Name Columns from the employees Table

UPDATING DATA IN A TABLE

Table data can be updated using the SQL *update* statement. Study the following example:

```
update employees set middle_name = "W" where employee_id = 1;
```

Figure 21-27 shows the results of the update statement by querying the employees table with the select statement.

JOINING TABLES WITH A SELECT STATEMENT

Two or more related tables can be joined to enhance data analysis. Since the employees table and the employee_training table are related via the employee_id attribute they can be joined in a query. However, before testing this SQL feature you should populate both the employees and employee_training tables with additional data. Fig-

```

mysql> select * from employees;
+-----+-----+-----+-----+-----+-----+
| EMPLOYEE_ID | FIRST_NAME | MIDDLE_NAME | LAST_NAME | SSN          | DATE_OF_BIRTH |
+-----+-----+-----+-----+-----+-----+
|          1 | Homer     | W           | Simpson  | 123-34-2345 | 1980-09-21    |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>

```

Figure 21-28: Results of the Update Statement — Note the Middle_Name is Changed to ‘W’
 Figures 21-29 and 21-30 show the data contents of the employees and employee_training tables respectively that will be used to illustrate the concepts in this section.

```

mysql> select * from employees;
+-----+-----+-----+-----+-----+-----+
| EMPLOYEE_ID | FIRST_NAME | MIDDLE_NAME | LAST_NAME | SSN          | DATE_OF_BIRTH |
+-----+-----+-----+-----+-----+-----+
|          1 | Homer     | W           | Simpson  | 123-34-2345 | 1980-09-21    |
|          2 | Steven    | Leroy       | Jones    | 111-11-1111 | 1963-01-31    |
|          3 | Harry     | Shim        | Potter   | 222-22-2222 | 1984-02-04    |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.01 sec)

mysql>

```

Figure 21-29: Employees Table with Additional Data Added

```

mysql> select * from employee_training;
+-----+-----+-----+-----+-----+
| EMPLOYEE_TRAINING_ID | EMPLOYEE_ID | DATE          | TOPIC                | RESULT |
+-----+-----+-----+-----+-----+
|          1           |          1   | 2004-06-05   | Nuclear plant management | failed |
|          2           |          1   | 2004-07-21   | Nuclear plant management | failed |
|          3           |          1   | 2004-08-10   | Nuclear plant management | failed |
|          4           |          1   | 2004-09-09   | Nuclear plant management | passed |
|          5           |          2   | 2003-10-30   | Wizard Ethics          | passed |
|          6           |          3   | 2005-01-05   | Computer Basics       | passed |
+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)

mysql>

```

Figure 21-30: Employee_Training Table Populated with Data

Referring to figures 21-29 and 21-30 – the employees table now contains two additional employees and the employee_training table has been populated with employee training data. Can you formulate the relationships between employees and their training by simply examining the data contained in the tables? Yes, in this simple example you can. However, with SQL you can perform a relational join on the tables using the *from* and *where* clauses of the select statement. Study the select statement shown in figure 21-31 along with its results.

```

mysql> select first_name, last_name, date, topic, result
-> from employees, employee_training
-> where employees.employee_id = employee_training.employee_id;
+-----+-----+-----+-----+-----+
| first_name | last_name | date          | topic                | result |
+-----+-----+-----+-----+-----+
| Homer     | Simpson  | 2004-06-05   | Nuclear plant management | failed |
| Homer     | Simpson  | 2004-07-21   | Nuclear plant management | failed |
| Homer     | Simpson  | 2004-08-10   | Nuclear plant management | failed |
| Homer     | Simpson  | 2004-09-09   | Nuclear plant management | passed |
| Steven    | Jones    | 2003-10-30   | Wizard Ethics          | passed |
| Harry     | Potter   | 2005-01-05   | Computer Basics       | passed |
+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)

mysql>

```

Figure 21-31: Results of Joining the employees Table with the employee_training Table

Referring to figure 21-31 — the select statement is used to extract meaningful data from both tables to answer the question: “What training courses has each employee taken?” The from clause specifies the two tables from which to

pull the data and the where clause specifies that the employees.employee_id value must equal the employee_training.employee_id value.

Let's pose another question to the database. "What training has Homer Simpson completed?" The resulting select statement looks like example 21.16.

21.16 Nested Select Statement

```

1  select first_name, last_name, date, topic, result
2  from employees, employee_training
3  where employees.employee_id = (select employee_id
4                                from employees
5                                where (first_name = "Homer" &&
6                                       last_name = "Simpson"))
7  = employee_training.employee_id;
```

Referring to example 21.16 — a nested select statement is used within the outer where clause to determine Homer Simpson's employee_id value. Notice the transitivity of the equality operator. Figure 21-32 shows the results of executing this select statement.

first_name	last_name	date	topic	result
Homer	Simpson	2004-06-05	Nuclear plant management	failed
Homer	Simpson	2004-07-21	Nuclear plant management	failed
Homer	Simpson	2004-08-10	Nuclear plant management	failed
Homer	Simpson	2004-09-09	Nuclear plant management	passed

Figure 21-32: Results of Executing the Nested Select Statement Shown in Example 21.16

ACCESSING CHAPTER_21 DATABASE TABLES VIA JDBC

The SQL statements demonstrated above can be applied to the chapter_21 database via a Java program that uses JDBC. As a quick review, the steps required to utilize JDBC in a Java program include 1) loading the database vendor's JDBC driver class using the Class.forName() method, 2) using the DriverManager.getConnection() method to establish a connection to the database, 3) using the connection object to create a statement or prepared statement object, 4) using the statement or prepared statement objects to execute SQL statements against the database, and 5) manipulating the data contained in the returned resultset object if applicable.

A SHORT JDBC EXAMPLE PROGRAM

The following short Java program executes these five steps to retrieve and display data contained in the chapter_21 database.

21.17 JDBCTestApp.java

```

1  import java.sql.*;
2
3  public class JDBCTestApp {
4      public static void main(String[] args){
5          try{
6              Class.forName("com.mysql.jdbc.Driver");
7              Connection conn = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/" +
8                                                           "chapter_21?user=swodog");
9              String default_query = "select * from employees";
10             Statement statement = conn.createStatement();
11             ResultSet rs = statement.executeQuery(default_query);
12             while(rs.next()){
13                 System.out.print(rs.getInt(1) + "\t\t");
14                 System.out.print(rs.getString(2) + "\t");
15                 System.out.print(rs.getString(3) + "\t");
16                 System.out.print(rs.getString(4) + "\t");
17                 System.out.print(rs.getString(5) + "\t");
18                 System.out.println(rs.getDate(6).toString());
19             }
20         }catch(Exception e){
21             e.printStackTrace();
22         }
23     }
24     } // end main() method
25 } // end JDBCTestApp class definition
```

Referring to example 21.17 — the `Class.forName()` method is used on line 6 to load the MySQL database driver class. The name of the MySQL driver class is “`com.mysql.jdbc.Driver`”. Next, the `DriverManager.getConnection()` method is used on lines 7 and 8 to create a `Connection` object. The `getConnection()` method requires a URL to the target database. In this example the URL to the `chapter_21` database located on the local host computer running on port 3306 connecting as user `swodog` is: “`jdbc:mysql://127.0.0.1:3306/chapter_21?user=swodog`”.

A default query string is formulated on line 9 and a `Statement` object is created on line 10 using the `Connection.createStatement()` method. Next, the `default_query` string is used as an argument to the `Statement.executeQuery()` method on line 11 to create a `ResultSet` object. Finally, the reference to the `ResultSet` object, `rs`, is used in the body of the while loop to access the returned query results. Figure 21-33 shows the results of running example 21.17.

```

Terminal — tcsh
[rick@millers-Computer:~/desktop/jdbc_test] swodog% java JDBCTestApp
1      Homer  W       Simpson 123-34-2345  1980-09-21
2      Steven Leroy  Jones  111-11-1111  1963-01-31
3      Harry  Shim   Potter  222-22-2222  1984-02-04
[rick@millers-Computer:~/desktop/jdbc_test] swodog%

```

Figure 21-33: Results of Running Example 21.17

ACCESSING RESULTSET METADATA

You can access descriptive data (*metadata*) about a `ResultSet` object by using its `getMetaData()` method. Accessing resultset metadata comes in handy for many reasons, especially if you want to determine table column name and type information. Example 21.18 is a slightly modified version of the `JDBCTestApp` program given in the previous example. The new version accesses and displays column name and type information for the employees table with the help of a `ResultSetMetaData` object.

21.18 `JDBCTestApp.java (mod 1)`

```

1      import java.sql.*;
2
3      public class JDBCTestApp {
4          public static void main(String[] args){
5              try{
6                  Class.forName("com.mysql.jdbc.Driver");
7                  Connection conn = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/" +
8                                                                "chapter_22?user=swodog");
9                  String default_query = "select * from employees";
10                 Statement statement = conn.createStatement();
11                 ResultSet rs = statement.executeQuery(default_query);
12                 ResultSetMetaData meta_data = rs.getMetaData();
13                 for(int i = 1; i <= meta_data.getColumnCount(); i++){
14                     System.out.print(meta_data.getColumnName(i) + ':' +
15                                     meta_data.getColumnTypeName(i) + '\t');
16                 }
17                 System.out.println();
18                 while(rs.next()){
19                     System.out.print(rs.getInt(1) + "\t\t");
20                     System.out.print(rs.getString(2) + "\t");
21                     System.out.print(rs.getString(3) + "\t");
22                     System.out.print(rs.getString(4) + "\t");
23                     System.out.print(rs.getString(5) + "\t");
24                     System.out.println(rs.getDate(6).toString());
25                 }
26             }catch(Exception e){
27                 e.printStackTrace();
28             }
29         }
30     } // end main() method
31 } // end JDBCTestApp class definition

```

Referring to example 21.18 — on line 12 a `ResultSetMetaData` reference named `meta_data` is declared and initialized with the help of the `ResultSet.getMetaData()` method. The for statement on line 13 prints the column name and column type name for each column in the employees table. This information is displayed above the employee data as is shown in figure 21-34.

```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/jdbc_test] swodog% java JDBCTestApp
EMPLOYEE_ID:INTEGER  FIRST_NAME:VARCHAR  MIDDLE_NAME:VARCHAR  LAST_NAME:VARCHAR  SSN:VARCHAR  DATE_OF_BIRTH:DATE
1      Homer  M      Simpson 123-34-2345  1988-09-21
2      Steven Leroy Jones 111-11-1111  1963-01-31
3      Harry  Shim  Potter 222-22-2222  1984-02-04
[Rick-Millers-Computer:~/desktop/jdbc_test] swodog%
    
```

Figure 21-34: Results of Running Example 21.18 with employee Table Metadata Displayed

Quick Review

MySQL is a relational database server. Before connecting to a MySQL database with JDBC you must install it, create a database, create database tables, and establish the necessary database access permissions. MySQL access control is essentially managed with the use of five tables located in the mysql database: user, db, host, tables_priv, and columns_priv. To alleviate complexity, database security can be ignored in a learning environment, however, in a production environment it is a subject of which you must be keenly aware.

The general steps to employing JDBC include 1) loading the database vendor’s JDBC driver class using the class.forName() method, 2) using the DriverManager.getConnection() method to establish a connection to the database, 3) using the connection object to create a statement or prepared statement object, 4) using the statement or prepared statement objects to execute SQL statements against the database, and 5) manipulating the data contained in the returned resultset object if applicable.

EXTENDED APPLET & JDBC EXAMPLE

It’s time now to tie all the concepts presented in this chapter together into one application. Let’s take a look again at the overall architectural diagram for the employee training management system.

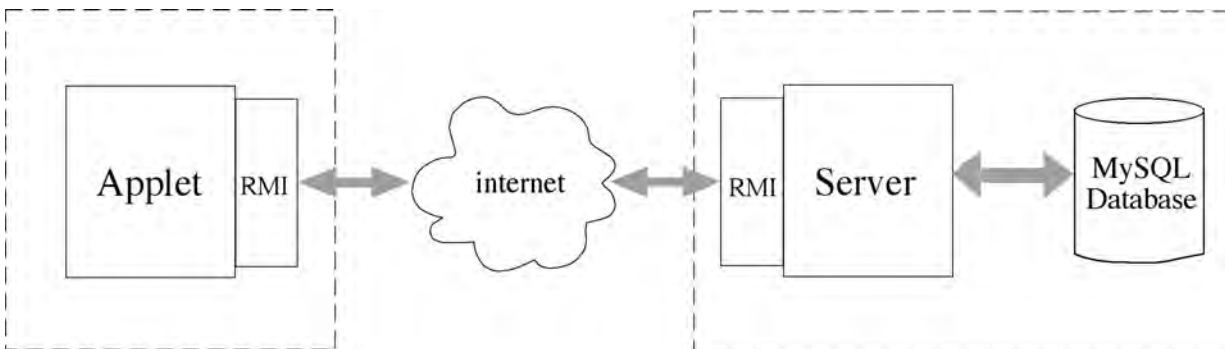


Figure 21-35: Employee Training Management System Architecture

Referring to figure 21-35 — an applet communicates with a server-based RMI component. The RMI method calls are translated into JDBC calls to the MySQL database. You are asked to refer to figures 21-13 and 21-14 for the detailed class diagrams for the server-side and client-side components respectively.

The Code

I have chosen to place the code for the employee training management system in the package structure depicted in figure 21-36. The completed application consists of nine classes arranged in four packages. Each class and a description of its purpose is listed in table 21-2.

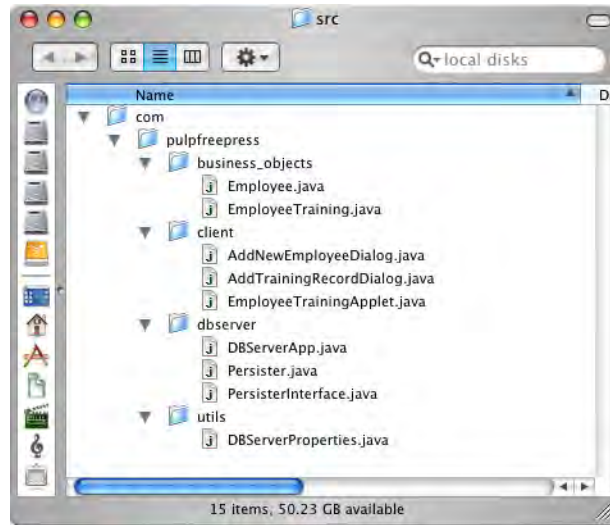


Figure 21-36: Employee Training Management System Source Code Package Structure

Class Name	Package	Description
Employee	com.pulpfreepress.business_objects	Java representation of an Employee entity.
EmployeeTraining	com.pulpfreepress.business_objects	Java representation of an EmployeeTraining entity.
AddNewEmployeeDialog	com.pulpfreepress.client	A dialog used by the EmployeeTrainingApplet class that lets users create a new employee.
AddTrainingRecordDialog	com.pulpfreepress.client	A dialog used by the EmployeeTrainingApplet class that lets users create a new employee training record.
EmployeeTrainingApplet	com.pulpfreepress.client	The main applet class for the client-side portion of the employee training management system.
DBServerApp	com.pulpfreepress.dbserver	The main application class for the server-side portion of the employee training management system. The DBServerApp class starts an instance of the RMI registry and registers an instance of the Persister class.
Persister	com.pulpfreepress.dbserver	The server-side RMI component that provides the touch point between clients and the MySQL database. Implements PersisterInterface.
PersisterInterface	com.pulpfreepress.dbserver	Specifies four remote methods.
DBServerProperties	com.pulpfreepress.utils	Extends the Properties class. Provides for flexible application configuration management by allowing application properties to be set via a properties file.

Table 21-2: Employee Training Management System Class Descriptions

21.19 Employee.java

```

1      package com.pulpfreepress.business_objects;
2
3      import java.util.*;
4      import java.io.*;
5
6      public class Employee implements Serializable {
7
8          private int      _id          = 0;
9          private String   _first_name  = null;
10         private String   _middle_name = null;
11         private String   _last_name   = null;
12         private String   _ssn         = null;
13         private String   _dob         = null;
14         private Vector   _child_relations = null;
15
16         public Employee(int id, String fn, String mn, String ln, String ssn, String dob, Vector cr){
17             _id = id;
18             _first_name = fn;
19             _middle_name = mn;
20             _last_name = ln;
21             _ssn = ssn;
22             _dob = dob;
23             if(cr != null) _child_relations = (Vector) cr.clone();
24         }
25
26         public Employee(){
27             this(0, null, null, null, null, null, null);
28         }
29
30         public void setEmployeeID(int id){ _id = id; }
31
32         public void setFirstName(String fn){ _first_name = fn; }
33
34         public void setMiddleName(String mn) { _middle_name = mn; }
35
36         public void setLastName(String ln) { _last_name = ln; }
37
38         public void setSSN(String ssn) { _ssn = ssn; }
39
40         public void setDOB(String dob) { _dob = dob; }
41
42         public void setChildRelations(Vector cr){ _child_relations = (Vector) cr.clone(); }
43
44         public int getEmployeeID() { return _id; }
45
46         public String getFirstName() { return _first_name; }
47
48         public String getMiddleName() { return _middle_name; }
49
50         public String getLastName() { return _last_name; }
51
52         public String getSSN() { return _ssn; }
53
54         public String getDOB() { return _dob; }
55
56         public Vector getChildRelations() { return (Vector) _child_relations.clone(); }
57     }
58

```

21.20 EmployeeTraining.java

```

1      package com.pulpfreepress.business_objects;
2
3      import java.io.*;
4
5      public class EmployeeTraining implements Serializable {
6
7          private int      _id          = 0;
8          private int      _emp_id      = 0;
9          private String   _date        = null;
10         private String   _topic        = null;
11         private String   _result       = null;
12
13         public EmployeeTraining(int id, int emp_id, String date, String topic, String result){
14             _id = id;
15             _emp_id = emp_id;
16             _date = date;
17             _topic = topic;

```

```

18     _result = result;
19     }
20
21     public EmployeeTraining(){
22         this(0, 0, null, null, null);
23     }
24
25     public void setEmployeeTrainingID(int id) { _id = id; }
26
27     public void setEmployeeID(int id) { _emp_id = id; }
28
29     public void setDate(String date){ _date = date; }
30
31     public void setTopic(String topic) { _topic = topic; }
32
33     public void setResult(String result) { _result = result; }
34
35     public int getEmployeeTrainingID() { return _id; }
36
37     public int getEmployeeID() { return _emp_id; }
38
39     public String getDate() { return _date; }
40
41     public String getTopic() { return _topic; }
42
43     public String getResult() { return _result; }
44
45     }

```

21.21 AddNewEmployeeDialog.java

```

1     package com.pulpfreepress.client;
2
3     import javax.swing.*;
4     import java.awt.event.*;
5     import java.awt.*;
6
7     public class AddNewEmployeeDialog extends JDialog {
8         private JLabel _label1 = null;
9         private JLabel _label2 = null;
10        private JLabel _label3 = null;
11        private JLabel _label4 = null;
12        private JLabel _label5 = null;
13        private JTextField _textfield1 = null;
14        private JTextField _textfield2 = null;
15        private JTextField _textfield3 = null;
16        private JTextField _textfield4 = null;
17        private JTextField _textfield5 = null;
18        private JButton _button1 = null;
19        private JPanel _panel1 = null;
20        private JPanel _blank_panel = null;
21
22        public AddNewEmployeeDialog(ActionListener al){
23            super(new JFrame(), "Add New Employee Dialog", true);
24
25            _label1 = new JLabel("First Name: ");
26            _label2 = new JLabel("Middle Name: ");
27            _label3 = new JLabel("Last Name: ");
28            _label4 = new JLabel("SSN (nnn-nn-nnnn): ");
29            _label5 = new JLabel("Date of Birth (yyyy-mm-dd): ");
30            _textfield1 = new JTextField(20);
31            _textfield2 = new JTextField(20);
32            _textfield3 = new JTextField(20);
33            _textfield4 = new JTextField(20);
34            _textfield5 = new JTextField(20);
35            _button1 = new JButton("Submit New Employee");
36            _button1.addActionListener(al);
37            _panel1 = new JPanel();
38            _panel1.add(_button1);
39            _blank_panel = new JPanel();
40
41            this.getContentPane().setLayout(new GridLayout(6, 2, 0, 0));
42            this.getContentPane().add(_label1);
43            this.getContentPane().add(_textfield1);
44            this.getContentPane().add(_label2);
45            this.getContentPane().add(_textfield2);
46            this.getContentPane().add(_label3);
47            this.getContentPane().add(_textfield3);
48            this.getContentPane().add(_label4);
49            this.getContentPane().add(_textfield4);
50            this.getContentPane().add(_label5);

```

```

51         this.getContentPane().add(_textfield5);
52         this.getContentPane().add(_blank_panel);
53         this.getContentPane().add(_panell);
54         this.pack();
55         this.setSize(400, 200);
56         this.setLocation(150, 150);
57     } // end constructor method
58
59     public void clearAllFields(){
60         _textfield1.setText("");
61         _textfield2.setText("");
62         _textfield3.setText("");
63         _textfield4.setText("");
64         _textfield5.setText("");
65     }
66
67     public String getFirstName() { return _textfield1.getText(); }
68     public String getMiddleName() { return _textfield2.getText(); }
69     public String getLastName() { return _textfield3.getText(); }
70     public String getSSN() { return _textfield4.getText(); }
71     public String getDOB() { return _textfield5.getText(); }
72
73 } // end AddNewEmployeeDialog class definition

```

21.22 AddTrainingRecordDialog.java

```

1     package com.pulpfreepress.client;
2
3     import javax.swing.*;
4     import java.awt.event.*;
5     import java.awt.*;
6
7     public class AddTrainingRecordDialog extends JDialog {
8         private JLabel _label1 = null;
9         private JLabel _label2 = null;
10        private JLabel _label3 = null;
11        private JTextField _textfield1 = null;
12        private JTextField _textfield2 = null;
13        private JTextField _textfield3 = null;
14        private JButton _button1 = null;
15        private JPanel _panell = null;
16        private JPanel _blank_panel = null;
17
18        public AddTrainingRecordDialog(ActionListener al){
19            super(new JFrame(), "Add Employee Training Record Dialog", true);
20            _panell = new JPanel();
21            _blank_panel = new JPanel();
22            _label1 = new JLabel("Date (yyyy-mm-dd):");
23            _label2 = new JLabel("Topic: ");
24            _label3 = new JLabel("Result: ");
25            _textfield1 = new JTextField(20);
26            _textfield2 = new JTextField(20);
27            _textfield3 = new JTextField(20);
28            _button1 = new JButton("Submit Training Record");
29            _button1.addActionListener(al);
30            _panell.add(_button1);
31            this.getContentPane().setLayout(new GridLayout(4, 2, 0, 0));
32            this.getContentPane().add(_label1);
33            this.getContentPane().add(_textfield1);
34            this.getContentPane().add(_label2);
35            this.getContentPane().add(_textfield2);
36            this.getContentPane().add(_label3);
37            this.getContentPane().add(_textfield3);
38            this.getContentPane().add(_blank_panel);
39            this.getContentPane().add(_panell);
40            this.setSize(300, 200);
41            this.setLocation(150, 150);
42            this.pack();
43        } // end constructor
44
45        public String getDate(){ return _textfield1.getText(); }
46        public String getTopic() { return _textfield2.getText(); }
47        public String getResult() { return _textfield3.getText(); }
48
49        public void clearAllFields(){
50            _textfield1.setText("");
51            _textfield2.setText("");
52            _textfield3.setText("");
53        }
54
55    } // end AddTrainingRecordDialog class definition

```

21.23 EmployeeTrainingApplet.java

```

1     package com.pulpfreepress.client;
2
3     import java.awt.*;
4     import java.awt.event.*;
5     import javax.swing.*;
6     import java.rmi.*;
7     import java.util.*;
8     import com.pulpfreepress.dbserver.*;
9     import com.pulpfreepress.business_objects.*;
10    import com.pulpfreepress.utils.*;
11
12    public class EmployeeTrainingApplet extends JApplet implements ActionListener {
13
14        private JPanel _panel1 = null;
15        private JPanel _panel2 = null;
16        private JPanel _south_panel = null;
17        private JTextArea _textareal = null;
18        private JScrollPane _scrollpanel = null;
19        private JButton _button1 = null;
20        private JButton _button2 = null;
21        private JButton _button3 = null;
22        private JButton _button4 = null;
23        private JTextField _textfield1 = null;
24        private JLabel _label1 = null;
25        private PersisterInterface _persister = null;
26        private Vector _employees = null;
27        private AddNewEmployeeDialog _add_employee_dialog = null;
28        private AddTrainingRecordDialog _add_training_record_dialog = null;
29        private static final int TEXTAREA_ROWS = 20;
30        private static final int TEXTAREA_COLS = 50;
31
32        public void init(){
33            initializeGui();
34            initializeRMI();
35            initializeDialogs();
36        }
37
38
39        private void initializeGui(){
40            _panel1 = new JPanel();
41            _panel2 = new JPanel();
42            _south_panel = new JPanel();
43            _textareal = new JTextArea(TEXTAREA_ROWS, TEXTAREA_COLS);
44            _textareal.setEditable(false);
45            _scrollpanel = new JScrollPane(_textareal);
46            _button1 = new JButton("Search By Last Name");
47            _button1.addActionListener(this);
48            _button2 = new JButton("Add New Employee");
49            _button2.addActionListener(this);
50            _button3 = new JButton("Get Employee Training Records");
51            _button3.addActionListener(this);
52            _button3.setEnabled(false);
53            _button4 = new JButton("Add Employee Training Record");
54            _button4.addActionListener(this);
55            _button4.setEnabled(false);
56            _textfield1 = new JTextField(20);
57            _label1 = new JLabel("Last Name: ");
58            _panel1.add(_label1);
59            _panel1.add(_textfield1);
60            _panel1.add(_button1);
61            _panel2.add(_button2);
62            _panel2.add(_button3);
63            _panel2.add(_button4);
64            _south_panel.setLayout(new GridLayout(2, 1, 0, 0));
65            _south_panel.add(_panel1);
66            _south_panel.add(_panel2);
67            this.getContentPane().add(_scrollpanel);
68            this.getContentPane().add(BorderLayout.SOUTH, _south_panel);
69        }
70
71        private void initializeRMI(){
72            try{
73                System.out.println("Attempting to lookup Employee_Persister_Service on : " +
74                    this.getCodeBase().getHost());
75                _persister = (PersisterInterface)Naming.lookup("rmi://" + this.getCodeBase().getHost() +
76                    "/Employee_Persister_Service");
77            }catch(Exception e){
78                e.printStackTrace();

```

```

79     }
80   }
81
82   private void initializeDialogs(){
83     _add_employee_dialog = new AddNewEmployeeDialog(this);
84     _add_employee_dialog.setVisible(false);
85     _add_training_record_dialog = new AddTrainingRecordDialog(this);
86     _add_training_record_dialog.setVisible(false);
87   }
88
89
90   public void actionPerformed(ActionEvent ae){
91     if(ae.getActionCommand().equals("Search By Last Name")){
92       try{
93         _employees = _persister.queryByLastName(_textfield1.getText());
94         if(_employees.size() > 0){
95           _button3.setEnabled(true);
96           _button4.setEnabled(true);
97         }
98         _textareal.setText("");
99         for(int i = 0; i < _employees.size(); i++){
100          _textareal.append(((Employee)_employees.elementAt(i)).getFirstName() + '\t' +
101                          ((Employee)_employees.elementAt(i)).getMiddleName() + '\t' +
102                          ((Employee)_employees.elementAt(i)).getLastName() + '\t' +
103                          ((Employee)_employees.elementAt(i)).getSSN() + '\n');
104        }
105      }catch(Exception e){
106        e.printStackTrace();
107      }
108    } else if(ae.getActionCommand().equals("Add New Employee")){
109      _add_employee_dialog.clearAllFields();
110      _add_employee_dialog.setVisible(true);
111    } else if(ae.getActionCommand().equals("Submit New Employee")){
112      try{
113        _add_employee_dialog.setVisible(false);
114        Employee emp = new Employee(0,
115                                   _add_employee_dialog.getFirstName(),
116                                   _add_employee_dialog.getMiddleName(),
117                                   _add_employee_dialog.getLastName(),
118                                   _add_employee_dialog.getSSN(),
119                                   _add_employee_dialog.getDOB(),
120                                   null);
121        _persister.addNewEmployee(emp);
122      }catch(Exception e){
123        e.printStackTrace();
124      }
125    }
126    } else if(ae.getActionCommand().equals("Get Employee Training Records")){
127      try{
128        _button3.setEnabled(false);
129        _button4.setEnabled(false);
130        Employee emp = (Employee)_employees.elementAt(getSelectedLineNumber());
131        emp = _persister.getEmployeeTrainingRecords(emp);
132        _textareal.setText("");
133        _textareal.append(emp.getFirstName() + '\t' +
134                          emp.getMiddleName() + '\t' +
135                          emp.getLastName() + '\t' +
136                          emp.getSSN() + '\n');
137        _textareal.append("-----\n");
138        Vector training_records = emp.getChildRelations();
139        for(int i = 0; i < training_records.size(); i++){
140          _textareal.append(((EmployeeTraining)training_records.elementAt(i)).getDate() + '\t' +
141                          ((EmployeeTraining)training_records.elementAt(i)).getTopic() + '\t' +
142                          ((EmployeeTraining)training_records.elementAt(i)).getResult() + '\n');
143        }
144      }catch(ArrayIndexOutOfBoundsException aioobe){
145        System.out.println("Invalid Employee Selection");
146        return;
147      }
148    } catch(Exception e){
149      e.printStackTrace();
150    }
151  } else if(ae.getActionCommand().equals("Add Employee Training Record")){
152    _add_training_record_dialog.clearAllFields();
153    _add_training_record_dialog.setVisible(true);
154  } else if(ae.getActionCommand().equals("Submit Training Record")){
155    try{
156      _add_training_record_dialog.setVisible(false);
157      Employee emp = (Employee)_employees.elementAt(getSelectedLineNumber());
158      EmployeeTraining et = new EmployeeTraining(0,

```

```

160                                     emp.getEmployeeID(),
161                                     _add_training_record_dialog.getDate(),
162                                     _add_training_record_dialog.getTopic(),
163                                     _add_training_record_dialog.getResult());
164         _persistor.addTrainingRecord(et);
165     }catch(ArrayIndexOutOfBoundsException aioobe){
166         System.out.println("Invalid employee selection");
167         return;
168     }
169     catch(Exception e){
170         e.printStackTrace();
171     }
172 }
173 }
174 } // end actionPerformed() method
175
176
177 private int getSelectedLineNumber(){
178     int caret_position = _textareal.getCaretPosition();
179     int line_number = 0;
180     int string_length = 0;
181     try{
182         StringTokenizer st = new StringTokenizer(_textareal.getText(), "\n");
183         while(caret_position >= (string_length += st.nextToken().length()) ){
184             line_number++;
185         }
186     }catch(NoSuchElementException ignored){ }
187     return line_number;
188 }
189
190 } // end EmployeeTrainingApplet class definition

```

21.24 DBServerApp.java

```

1     package com.pulpfreepress.dbserver;
2
3     import com.pulpfreepress.utils.*;
4     import java.rmi.*;
5     import java.rmi.registry.*;
6
7     public class DBServerApp {
8         public static void main(String[] args){
9             DBServerProperties props = DBServerProperties.getInstance();
10            try{
11                System.out.println("Starting registry...");
12                LocateRegistry.createRegistry(Integer.parseInt(props.getProperty(props.DEFAULT_RMI_PORT)));
13                System.out.println("Registry started on port " + props.getProperty(props.DEFAULT_RMI_PORT));
14                System.out.println("Binding Persister service name with Persister instance...");
15                Naming.bind(props.getProperty(props.DEFAULT_EMPLOYEE_PERSISTER_SERVICE), new Persister());
16                System.out.println("Bind successful.");
17                System.out.println("Ready for remote method invocation.");
18            }catch(Exception e){
19                e.printStackTrace();
20            }
21        } // end main() method
22    } // end DBServerApp class definition

```

21.25 Persister.java

```

1     package com.pulpfreepress.dbserver;
2
3     import java.sql.*;
4     import java.util.*;
5     import java.rmi.*;
6     import java.rmi.server.*;
7     import com.pulpfreepress.business_objects.*;
8     import com.pulpfreepress.utils.*;
9
10
11    public class Persister extends UnicastRemoteObject implements PersisterInterface {
12        private static DBServerProperties _props = null;
13        private Connection _conn = null;
14
15        // Employees Table Column Values
16        private static final int ID_COL = 1;
17        private static final int FN_COL = 2;
18        private static final int MN_COL = 3;
19        private static final int LN_COL = 4;
20        private static final int SSN_COL = 5;
21        private static final int DOB_COL = 6;
22

```

```

23     // EmployeeTraining Table Column Values
24     private static final int TR_ID_COL = 1;
25     private static final int EMP_ID_COL = 2;
26     private static final int DATE_COL = 3;
27     private static final int TOPIC_COL = 4;
28     private static final int RESULT_COL = 5;
29
30
31     static{
32         _props = DBServerProperties.getInstance();
33     }
34
35     public Persister() throws RemoteException {
36         try{
37             System.out.println("Loading JDBC driver class...");
38             Class.forName(_props.getProperty(_props.DEFAULT_JDBC_DRIVER));
39             System.out.println("JDBC driver class loaded successfully.");
40             System.out.println("Creating connection to database...");
41             _conn = DriverManager.getConnection(_props.getProperty(_props.DEFAULT_MYSQL_JDBC_PROTOCOL) +
42                 "://" +
43                 _props.getProperty(_props.DEFAULT_SERVER_IP) +
44                 ":" +
45                 _props.getProperty(_props.DEFAULT_MYSQL_DB_PORT) +
46                 "/" +
47                 _props.getProperty(_props.DEFAULT_DATABASE) +
48                 "?user=" +
49                 _props.getProperty(_props.DEFAULT_DB_USER));
50             System.out.println("Database connection created successfully.");
51             System.out.println("Persister object created successfully.");
52         }catch(Exception e){
53             e.printStackTrace();
54         }
55     }
56     } // end constructor
57
58
59     public void finalize() throws Throwable {
60         _conn.close();
61         super.finalize();
62     }
63
64     public synchronized Vector queryByLastName(String last_name){
65         Vector employee_vector = new Vector();
66         Statement statement = null;
67         ResultSet result_set = null;
68         String query = null;
69         try{
70             if((last_name == null) || (last_name.equals(""))){
71                 query = "SELECT * FROM employees";
72                 statement = _conn.createStatement();
73                 result_set = statement.executeQuery(query);
74             }else{
75                 query = "SELECT * FROM employees WHERE (employees.last_name = '" + last_name + "')";
76                 statement = _conn.createStatement();
77                 result_set = statement.executeQuery(query);
78             }
79
80             while(result_set.next()){
81                 employee_vector.add(new Employee(result_set.getInt(ID_COL),
82                     result_set.getString(FN_COL),
83                     result_set.getString(MN_COL),
84                     result_set.getString(LN_COL),
85                     result_set.getString(SSN_COL),
86                     result_set.getDate(DOB_COL).toString(),
87                     null));
88             }
89         }catch(Exception e){
90             e.printStackTrace();
91         }
92         finally{
93             try{
94                 if(statement != null) statement.close();
95                 if(result_set != null) result_set.close();
96                 System.out.println("statement & result_set closed");
97             }catch(Exception ignored){ }
98         }
99         return employee_vector;
100     } // end queryByLastName() method
101
102
103

```

```

104     public synchronized void addNewEmployee(Employee emp){
105         PreparedStatement statement = null;
106         String insert =
107             "INSERT INTO employees (employee_id, first_name, middle_name, last_name, ssn, date_of_birth)" +
108             "VALUES (?, ?, ?, ?, ?, ?)";
109         try{
110             statement = _conn.prepareStatement(insert);
111             statement.setInt(ID_COL, 0);
112             statement.setString(FN_COL, emp.getFirstName());
113             statement.setString(MN_COL, emp.getMiddleName());
114             statement.setString(LN_COL, emp.getLastName());
115             statement.setString(SSN_COL, emp.getSSN());
116             statement.setDate(DOB_COL, java.sql.Date.valueOf(emp.getDOB()));
117             statement.executeUpdate();
118         }catch(Exception e){
119             e.printStackTrace();
120         }
121         finally{
122             try{
123                 statement.close();
124             }catch(Exception ignored){ }
125         }
126     } // end addNewEmployee() method
127
128
129     public synchronized void addTrainingRecord(EmployeeTraining et){
130         PreparedStatement statement = null;
131         String insert =
132             "INSERT INTO employee_training (employee_training_id, employee_id, date, topic, result)" +
133             "VALUES (?, ?, ?, ?, ?)";
134         try{
135             statement = _conn.prepareStatement(insert);
136             statement.setInt(TR_ID_COL, 0);
137             statement.setInt(EMP_ID_COL, et.getEmployeeID());
138             statement.setDate(DATE_COL, java.sql.Date.valueOf(et.getDate()));
139             statement.setString(TOPIC_COL, et.getTopic());
140             statement.setString(RESULT_COL, et.getResult());
141             statement.executeUpdate();
142
143         }catch(Exception e){
144             e.printStackTrace();
145         }
146         finally{
147             try{
148                 statement.close();
149             }catch(Exception ignored){ }
150         }
151     } // end addTrainingRecord() method
152
153     public synchronized Employee getEmployeeTrainingRecords(Employee emp){
154         PreparedStatement statement = null;
155         ResultSet result_set = null;
156         Vector training_records = new Vector();
157         String query = "SELECT * " +
158             "FROM employee_training " +
159             "WHERE (employee_training.employee_id = ?)";
160         try{
161             statement = _conn.prepareStatement(query);
162             statement.setInt(1, emp.getEmployeeID());
163             result_set = statement.executeQuery();
164             while(result_set.next()){
165                 training_records.add(new EmployeeTraining(result_set.getInt(TR_ID_COL),
166                                                             result_set.getInt(EMP_ID_COL),
167                                                             result_set.getDate(DATE_COL).toString(),
168                                                             result_set.getString(TOPIC_COL),
169                                                             result_set.getString(RESULT_COL)));
170             }
171             emp.setChildRelations(training_records);
172
173         }catch(Exception e){
174             e.printStackTrace();
175         }
176         finally{
177             try{
178                 statement.close();
179                 result_set.close();
180             }catch(Exception ignored){ }
181         }
182         return emp;
183     } // end getEmployeeTrainingRecords() method
184

```



```

185
186
187     } // end Persister class definition

```

21.26 DBServerProperties.java

```

1     package com.pulpfreepress.utils;
2
3     import java.util.*;
4     import java.io.*;
5
6
7     public class DBServerProperties extends Properties {
8
9         // class constants - default key strings
10        public static final String DEFAULT_RMI_PORT = "DEFAULT_RMI_PORT";
11        public static final String DEFAULT_SERVER_IP = "DEFAULT_SERVER_IP";
12        public static final String DEFAULT_PROPERTIES_FILE = "DEFAULT_PROPERTIES_FILE";
13        public static final String DEFAULT_JDBC_DRIVER = "DEFAULT_JDBC_DRIVER";
14        public static final String DEFAULT_MYSQL_DB_PORT = "DEFAULT_MYSQL_DB_PORT";
15        public static final String DEFAULT_DATABASE = "DEFAULT_DATABASE";
16        public static final String DEFAULT_DB_USER = "DEFAULT_DB_USER";
17        public static final String DEFAULT_MYSQL_JDBC_PROTOCOL = "DEFAULT_MYSQL_JDBC_PROTOCOL";
18        public static final String DEFAULT_EMPLOYEE_PERSISTER_SERVICE =
19            "DEFAULT_EMPLOYEE_PERSISTER_SERVICE";
20
21
22        // class constants - default value strings
23        private static final String DEFAULT_RMI_PORT_VALUE = "1099";
24        private static final String DEFAULT_SERVER_IP_VALUE = "127.0.0.1";
25        private static final String DEFAULT_PROPERTIES_FILE_VALUE = "DBserver.properties";
26        private static final String DEFAULT_JDBC_DRIVER_VALUE = "com.mysql.jdbc.Driver";
27        private static final String DEFAULT_MYSQL_DB_PORT_VALUE = "3306";
28        private static final String DEFAULT_DATABASE_VALUE = "chapter_21";
29        private static final String DEFAULT_DB_USER_VALUE = "swodog";
30        private static final String DEFAULT_MYSQL_JDBC_PROTOCOL_VALUE = "jdbc:mysql";
31        private static final String DEFAULT_EMPLOYEE_PERSISTER_SERVICE_VALUE =
32            "Employee_Persister_Service";
33
34        // class variables
35        private static DBServerProperties _properties_object = null;
36
37
38
39        private DBServerProperties( String properties_file ){
40            try{
41                FileInputStream fis = new FileInputStream(properties_file);
42                load(fis);
43            }catch(Exception e) {
44                System.out.println("Problem opening properties file!");
45                System.out.println("Bootstrapping properties...");
46                try{
47                    FileOutputStream fos = new FileOutputStream(DEFAULT_PROPERTIES_FILE_VALUE);
48                    setProperty(DEFAULT_RMI_PORT, DEFAULT_RMI_PORT_VALUE);
49                    setProperty(DEFAULT_SERVER_IP, DEFAULT_SERVER_IP_VALUE);
50                    setProperty(DEFAULT_PROPERTIES_FILE, DEFAULT_PROPERTIES_FILE_VALUE);
51                    setProperty(DEFAULT_JDBC_DRIVER, DEFAULT_JDBC_DRIVER_VALUE);
52                    setProperty(DEFAULT_MYSQL_DB_PORT, DEFAULT_MYSQL_DB_PORT_VALUE);
53                    setProperty(DEFAULT_DATABASE, DEFAULT_DATABASE_VALUE);
54                    setProperty(DEFAULT_DB_USER, DEFAULT_DB_USER_VALUE);
55                    setProperty(DEFAULT_MYSQL_JDBC_PROTOCOL, DEFAULT_MYSQL_JDBC_PROTOCOL_VALUE);
56                    setProperty(DEFAULT_EMPLOYEE_PERSISTER_SERVICE, DEFAULT_EMPLOYEE_PERSISTER_SERVICE_VALUE);
57
58                    super.store(fos, "DBServerProperties File - Edit Carefully");
59                    fos.close();
60                }catch(Exception e2){ System.out.println("Uh ohh...Bigger problems exist!"); }
61            }
62        }
63
64
65        /*****
66        * Private default constructor. Applications will get an instance via the getInstance() method.
67        * @see getInstance()
68        *****/
69        private DBServerProperties(){
70            this(DEFAULT_PROPERTIES_FILE_VALUE);
71        }
72
73        /*****
74        * The store() method attempts to persist its properties collection.
75        *****/

```

```

76     public void store(){
77         try{
78             FileOutputStream fos = new
79                 FileOutputStream(getProperty(DEFAULT_PROPERTIES_FILE));
80             super.store(fos, "DBServerProperties File");
81             fos.close();
82         }catch(Exception e){ System.out.println("Trouble storing properties!"); }
83     }
84
85     /*****
86     *  getInstance() returns a singleton instance if the DBServerProperties object.
87     *****/
88
89     public static DBServerProperties getInstance(){
90         if(_properties_object == null){
91             _properties_object = new DBServerProperties();
92         }
93         return _properties_object;
94     }
95 } // end DBServerProperties class definition

```

21.27 employeetraining.html

```

1     <title>Employee Training Applet Page</title>
2     <hr>
3     <applet archive="EmployeeAppletClasses.jar"
4         code="com.pulpfreepress.client.EmployeeTrainingApplet.class"
5         width=650
6         height=400>
7     </applet>
8     <hr>

```

Compiling And Packaging The Applet Code For Distribution

Before the example code can be executed it must be compiled, the `Persister_Stub.class` must be generated with the `rmic` tool, the class files must be packaged into a jar file, and the HTML page shown in example 21.27 must be created.

Compiling The Source Code

The easiest way to compile the code is to compile everything at once. You can do this by changing to the directory that contains the `src` directory and issuing the `javac` in the format of the following example:

```
javac -d . src/com/pulpfreepress/*/*.java
```

The `-d .` option signals the `javac` compiler to place the resulting class files in the current working directory. They will be written there in their proper package structure. The `src/com/pulpfreepress/*/*.java` path specifies all the java source files contained in all the subdirectories located in the `src/com/pulpfreepress` directory. (**Note:** *This form of the `javac` command will not work in Windows! But it does work on Apple's OS X and on SuSE Linux 9.3.*)

GENERATING THE PERSISTER_STUB.CLASS FILE

After the code is compiled and the class files generated you must generate the `Persister_Stub.class` file using the `rmic` tool. From the project working directory enter the following command:

```
rmic -d . -v1.2 com.pulpfreepress.dbserver.Persister
```

Check to ensure the `Persister_Stub.class` file was generated.

CREATING THE JAR FILE

The classes must now be packaged in a jar file named `EmployeeAppletClasses.jar` since this is the archive file named in the archive attribute of the applet tag shown in example 21.27. To create the jar file use the jar utility from the working directory in the following fashion:

```
jar -cf EmployeeAppletClasses.jar com
```

The `-cf EmployeeAppletClasses.jar` tells the jar utility to create a file named `EmployeeAppletClasses.jar`. The `com` at the end is the name of the directory (*and the packages and classes it contains*) to archive.

RUNNING THE EXAMPLE

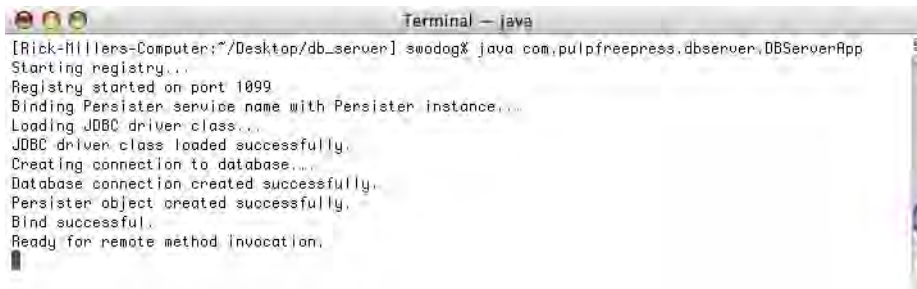
To run the example you must first make sure MySQL is up and running with the `chapter_21` database properly set up. Next, you must start the `DBServerApp` program on the server computer, and finally, access the `employeetraining.html` page locally with a browser or serve it up with a web server and access it remotely.

START MySQL DATABASE APPLICATION

If it's not already running start MySQL. The MySQL application can be set up to start automatically when the computer is powered up.

START THE DBSERVERAPP PROGRAM

After MySQL is running start the `DBServerApp` program. You should see an output on the terminal similar to figure 21-37.



```
Terminal - java
[rick-millers-Computer:~/Desktop/db_server] sudo java com.pulpfreepress.dbserver.DBServerApp
Starting registry...
Registry started on port 1099
Binding Persister service name with Persister instance...
Loading JDBC driver class...
JDBC driver class loaded successfully
Creating connection to database...
Database connection created successfully
Persister object created successfully
Bind successful
Ready for remote method invocation.
```

Figure 21-37: Terminal Output Showing `DBServerApp` Startup Sequence

ACCESS THE EMPLOYEETRaining.HTML PAGE

Before you do this make sure the `EmployeeAppletClasses.jar` file is placed in the same directory as the `employeetraining.html` page. Once you've done this start a browser, browse to the directory where the HTML page is located and open the page. Your browser window should look similar to figure 21-38.

USING THE EMPLOYEE TRAINING APPLET

To get a list of all employees simply click the Search By Last Name button leaving the Last Name text field blank. The results are shown in figure 21-39. To show an employee's training records click on an employee row in the text area and then click the Get Employee Training Records button. The results are shown in figure 21-40.

To add a new employee click the Add New Employee button to display the Add New Employee dialog. Figures 21-41 and 21-42 illustrate. Figure 21-43 shows a new list of all employees. New employee training records can be added by clicking on an employee and then clicking the Add Employee Training Record button. This is not shown.

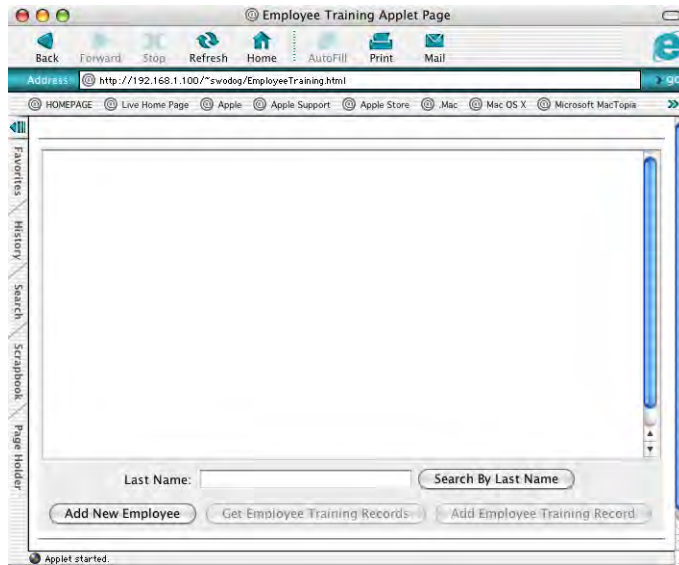


Figure 21-38: EmployeeTrainingApplet Appearance on First Access



Figure 21-39: Complete List of Employees

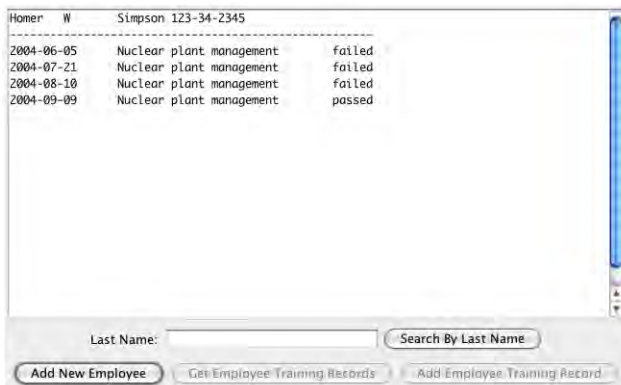


Figure 21-40: Training Records for Homer Simpson

Figure 21-41: Add New Employee Dialog

Figure 21-42: Add New Employee Dialog with Text Fields Filled In

Figure 21-43: New Employee Added to the Database

Code Discussion

This section presents a brief discussion of the employee training management system source code presented in examples 21.19 through 21.27. Only the highlights are discussed and you are encouraged to create sequence diagrams that trace code execution throughout the related classes.

Employee and EmployeeTraining Classes

Referring to examples 21.19 and 21.20 — the `Employee` and `EmployeeTraining` classes each represent a business object within the employee training management system application domain. The `Employee` class contains fields that represent important employee attributes. Additionally, on line 14 of example 21.19, the `Employee` class declares a `Vector` reference field named `_child_relations`. The purpose of the `_child_relations` field is to contain references to an employee's training records. This field is set to null and populated only when an employee's training records are retrieved from the database.

The `EmployeeTraining` class models the employee training entity. Each of its fields maps to a column in the `employee_training` database table.

AddNewEmployeeDialog AND AddTrainingRecordDialog Classes

Referring to examples 21.21 and 21.22 —these two dialog classes are utilized by the EmployeeTrainingApplet class to capture new employee and employee training information for entry into the database.

EmployeeTrainingApplet Class

Referring to example 21.23 — the EmployeeTrainingApplet class is the primary client-side component. Its purpose is to provide the primary user interface, initialize the RMI service, and translate GUI actions into RMI calls where required. The initializeRMI() method on line 71 is responsible for looking up the employee persister service (*Employee_Persister_Service*) on the host server.

The actionPerformed() method on line 90 handles all the button clicks, even for its two dialog classes. When the “Search By Last Name” button is clicked the persister’s queryByLastName() method is called on line 91 using the contents of _textfield1 as an argument. The _employees Vector reference is initialized as a result of the method call and if its length is greater than zero then each Employee object’s information is appended to the text area on lines 100 through 103. At this point the _child_relations field of each Employee object is not initialized.

When an employee is selected in the text area and the “Get Employee Training Records” button is clicked, the getSelectedLineNumber() method is called to determine which employee is selected. The Employee reference corresponding to the selected line number is retrieved from the _employees Vector and the persister’s getEmployeeTrainingRecords() method is called with the Employee reference as an argument on line 131. The employee’s training records are then displayed in the text area.

DBServerApp Class

Referring to example 21.24 — the DBServerApp class serves as the main application class for the server-side component of the employee training management system. Its primary job is to start the RMI registry and register an instance of the Persister class to its service name. Note that all information required by the DBServerApp class is provided by the DBServerProperties class and retrieved from a properties file.

Persister Class

Referring to example 21.25 — the Persister class is the primary server-side component. Its job is to translate RMI method calls into JDBC calls against the employee training management database. The JDBC initialization takes place in the constructor method beginning on line 35. This class, too, utilizes the DBServerProperties class to retrieve server-side configuration information.

Stepping through the queryByLastName() method on line 64 — the main action occurs in the try-catch block starting on line 69. If the last_name String reference is equal to null or contains no value then a query is formulated to retrieve all employees from the database. If, however, the last_name string contains a value, it is used to query on employees having that last name.

The while loop starting on line 80 processes the returned ResultSet and populates a Vector with Employee references. When all is complete the statement and result_set close() methods are called and the Vector of Employee references is returned.

Let’s look now at the getEmployeeTrainingRecords() method on line 153. An Employee reference serves as the method parameter. This Employee reference is already populated with all employee attribute data with the exception of its related employee training records. The Employee reference is used to extract the employeeID which is used to query the database for that employee’s training records. A Vector containing the employee’s training records is populated in the while loop starting on line 164. On line 171 the setChildRelations() method is called using the populated Vector as an argument. The fully populated Employee reference is then returned.

DBServerProperties

Referring to example 21.26 — the DBServerProperties class extends the java.util.Properties class and is utilized by the DBServerApp and Persister classes to retrieve application configuration information from a properties file. You’ve seen examples of Properties classes in previous chapters. I’d like to draw your attention to the default value

strings defined on lines 23 through 32. To get the employee training management system to run you will more than likely need to change the following default values: `DEFAULT_SERVER_IP_VALUE`, `DEFAULT_DATABASE_VALUE`, & `DEFAULT_DB_USER_VALUE`.

STATEMENTS VS. PREPARED STATEMENTS

The `Persister` class makes use of JDBC Statements and `PreparedStatement`s. (A third type of JDBC statement, `CallableStatement`, is used to call stored database procedures and is not utilized in this example.) All three types of JDBC statements can be retrieved from a JDBC Connection via its `createStatement()`, `prepareStatement()`, and `prepareCall()` methods respectively. I'd like to dwell for a moment on when you would use a `Statement` vs. a `PreparedStatement`.

STATEMENT

Use a `Statement` when you are executing a simple query statement (*select, from, where*). By simple I mean one that is not built from too many string concatenations. Long or complex string concatenations are inefficient and error prone.

PREPARED STATEMENT

Use a `PreparedStatement` when the query is complex or you are performing inserts and updates to the database. `PreparedStatement`s can be used repeatedly and are more efficient than their `Statement` counterparts.

A `PreparedStatement` is formulated with the help of variable placeholders in the form of question marks: '?'. Refer to the `addTrainingRecord()` method of the `Persister` class starting on line 129. An `EmployeeTraining` reference is passed as an argument to the method call. The SQL insert statement string is formulated on lines 131 through 133. The question marks that appear in the parentheses of the values clause on line 133 correspond, from left to right, to the fields that are listed between the parentheses on line 132. On line 135 the `PreparedStatement` is created using the insert string as an argument. However, before the prepared statement can be executed, each statement variable must be set using a set method that corresponds to the appropriate type. (i.e., `setInt()`, `setString()`, etc.) Only after all statement variables are properly set can the `PreparedStatement`'s `executeUpdate()` method be called.

SUMMARY

An applet is a Java program that can be embedded in a web page with an `<applet>` tag and run in a web browser. An applet extends the `JApplet` class and thus gains an enormous amount of functionality. Applets have methods that make it easy to manipulate sound and image files.

Applets have four life-cycle stages: initialization, start, stop, and destroy. Each stage calls the corresponding life-cycle method: `init()`, `start()`, `stop()`, and `destroy()`. Applets do not have to implement all life-cycle methods.

The `init()` method functions much like a constructor and is where applet state initialization code is usually placed. An applet can also have constructor methods if required, one of which must be a no-argument constructor.

Place clean-up code in the `destroy()` method.

Applets run in a restricted environment and are considered to be untrusted code. Applets cannot, as a rule, access the local file system or initiate or receive network connections to any computer other than the one from which it was served.

Use `<param>` tags to embed applet parameters in HTML pages. Use the `Applet` class's `getParameter()` method to fetch parameter values and use them in your program. Use wrapper classes as required to convert parameter value strings to primitive type values.

JDBC™ is an API that lets you access data sources from Java programs. The JDBC API comes in two packages: the `java.sql` package contains the JDBC core API and the `javax.sql` package contains the JDBC Optional Package API. The JDBC API consists mostly of interfaces which must be implemented by Java platform vendors. Database vendors must also supply JDBC driver classes to facilitate JDBC connectivity to their database products.

MySQL is a relational database server. Before connecting to a MySQL database with JDBC you must install it, create a database, create database tables, and establish the necessary database access permissions. MySQL access control is essentially managed with the use of five tables located in the mysql database: user, db, host, tables_priv, and columns_priv. To alleviate complexity, database security can be ignored in a learning environment, however, in a production environment it is a subject of which you must be keenly aware.

The general steps to employing JDBC include 1) loading the database vendor's JDBC driver class using the `Class.forName()` method, 2) using the `DriverManager.getConnection()` method to establish a connection to the database, 3) using the connection object to create a statement or prepared statement object, 4) using the statement or prepared statement objects to execute SQL statements against the database, and 5) manipulating the data contained in the returned resultset object if applicable.

Use JDBC Statements when dealing with simple queries. Otherwise use PreparedStatements for efficiency and ease of use.

Skill-Building Exercises

- Trace Applet Life Cycle Stages:** Run the BasicApplet program given in this chapter and trace the applet life cycle stages. Note the effects of first accessing the applet's web page, moving off the page, clicking the page refresh button, and exiting the browser. Open two browser windows, access the BasicApplet page, and trace the life cycle stage messages printed to the text area of each window. Can you explain what you see?
- <object> Tag:** Convert the applet examples in this chapter to use the <object> tag vs. the <applet> tag.
- API Research:** Research the two JDBC packages: `java.sql` and `javax.sql`. List and describe all classes, interfaces, and exceptions. Take note of each interface and class methods, their required parameters and return values.
- Directed Study:** Expand your knowledge of the Structured Query Language. Procure a reference on SQL and read it. How does MySQL's implementation of SQL compare with the SQL standard? What SQL language features, if any, does MySQL not implement?
- Applet Security:** Research browser security policy and the use of signed applets.
- Relational Database Design:** Procure and read a book about relational database design. (*Honestly, if you wait to take a class for everything you need to learn you will never catch up to the technology curve!*)
- Regular Expressions:** In their current state, the two dialog classes utilized by the EmployeeTrainingApplet of the employee training management system implement no user entry error checking. Study Java's implementation of regular expressions and modify the program to utilize regular expressions to ensure that only acceptable data is sent to the database for persistence.
- Programming Exercise:** Modify the Poetry applet so that the canvas and tile colors can be set via applet parameters.
- Programming Exercise:** Modify the employee training management system to use a JTable to display employee and employee training data.
- JDBC RowSet:** Research the JDBC RowSet. Modify the employee training management system to utilize RowSets.

SUGGESTED PROJECTS

1. **Stock Photograph Database:** Design and implement an application that allows users to browse and download digital images by different categories (i.e., subject, photographer, etc.). Utilize RMI for client-server communication, and MySQL as the database engine.
2. **Property Management Database:** Design and implement an application that allows property management professionals to manage their properties. The application must allow them to input new properties under their management along with one or more pictures of each property. New property information might include address, number of bedrooms, bathrooms and other related descriptive elements, amount of rent, owner and tenant information. Other things to consider: Owners can own more than one property. Tenants can rent more than one property. Allow users to browse available properties based on search criteria that might include address, zip code, number of bedrooms, etc.
3. **Web Research Database:** Design and implement an application that allows you to ingest URL links along with descriptive information to interesting sites you find while conducting research on the web.
4. **Personal Library Management System:** Design and implement an application that allows users to manage their personal library holdings regardless of media. The application should allow users to enter and manipulate information about each item and search the database for items by topic, author, artist, ISBN, etc.
5. **Home Inventory Management System:** Design and implement an application that allows users to manage their personal home inventories. The application should allow users to input and manipulate information about each item including original cost, replacement cost, description, model, brand, location, etc.
6. **Online Medical Records System:** Design and implement an application that allows physicians to enter and retrieve patient treatment information. (*For the purpose of this exercise keep this system simple as you can literally spend years designing this thing!*)
7. **Online Employee Performance Review System:** Design and implement an employee performance review system. The application must allow employees to start the review process by creating a new review and adding their inputs and achievements for the year. When they “submit” the review it must be made available to the employee’s supervisor so they can add comments and submit to corporate HR.
8. **Online Resume Submission System:** Design and implement an application that lets users submit text-based resumes online. The application would allow prospective employee users to paste an ASCII version of their resume into a text area. When they click submit the application would process the text area and extract keywords that can be inserted into a database along with the text of the resume. The application must allow company personnel to search the resume database for potential hires based on a keyword search.
9. **National Sex Offender Tracking Database:** Law enforcement needs your help! Your mission — design and implement a national sex offender registration and tracking database. This application will be used by law enforcement personnel to keep tabs on the riffraff and facilitate information sharing between agencies.
10. **National Handgun Purchase Authorization Database:** Your country needs your help yet again! To ensure customers truly have the right to keep and bear handguns their information must be submitted to an online application that searches a criminal history database. If there is a match they are denied the right to purchase.

SELF-TEST QUESTIONS

1. How is an applet different from an application?
2. List at least four applet security restrictions.
3. List and briefly discuss the four applet life-cycle states and their associated applet life-cycle methods.
4. (True/False) An applet can initiate network connections to, and accept incoming network connections from, any computer on the network.
5. List and describe the purpose of each <applet> tag attribute.
6. How can information be passed to an applet from an HTML page?
7. List and briefly discuss the general steps required to create a database in MySQL.
8. List and briefly discuss the steps required to utilize JDBC in a program.
9. What two packages contain the JDBC classes and interfaces?
10. Describe at least one benefit derived from using applets vs. applications.
11. What type of constructor, if it has any, must an applet have?
12. What additional code must be written before you can run an applet?
13. What's the difference between JDBC Statements and PreparedStatement?
14. What character is used as a variable placeholder in a PreparedStatement?
15. Describe a scenario where you would choose to use a PreparedStatement vs. a Statement.

REFERENCES

HTML 4.01 Specification [<http://www.w3.org/TR/html4/>]

Donald Bales, *JDBC™ Pocket Reference*, O'Reilly & Associates, Inc., Sebastopol, CA. ISBN: 0-596-00457-5.

George Reese, *MySQL Pocket Reference*, O'Reilly & Associates, Inc., Sebastopol, CA. ISBN: 0-596-00446-X

Randy Jay Yarger, et. al., *MySQL & mSQL*, O'Reilly & Associates, Inc., Sebastopol, CA. ISBN: 1-56592-434-7

Heikki Mannila, et. al., *The Design of Relational Databases*, Addison-Wesley, Reading, MA. ISBN: 0-201-56523-4

JDBC 3.0 Specification. Available at: [<http://java.sun.com/products/jdbc/download.html>]

JDBC 3.0 API Documentation. Available at: [<http://java.sun.com/products/jdbc/download.html>]

Rick F van der Lans, *Introduction to SQL*, Addison-Wesley, Reading, MA. ISBN: 0-201-17521-5

MySQL Connector/J Documentation. Available at: [www.mysql.com]

NOTES

PART VI: OBJECT-ORIENTED DESIGN

CHAPTER 22



Crowded City STREET

INHERITANCE, COMPOSITION, INTERFACES, POLYMORPHISM

LEARNING OBJECTIVES

- *LIST AND DISCUSS THE BENEFITS OFFERED BY THE USE OF INHERITANCE*
- *LIST AND DISCUSS THE BENEFITS OFFERED BY THE USE OF COMPOSITION*
- *DESCRIBE WHEN INHERITANCE IS AN APPROPRIATE DESIGN MECHANISM*
- *LIST THE THREE ESSENTIAL PURPOSES OF INHERITANCE*
- *LIST AND DESCRIBE THE INHERITANCE FORMS INCLUDED IN MEYER'S INHERITANCE TAXONOMY*
- *UTILIZE COAD'S FIVE INHERITANCE CHECKPOINTS TO DETERMINE THE EFFECTIVE USE OF INHERITANCE*
- *DESCRIBE THE PURPOSE OF AN INTERFACE*
- *STATE THE DEFINITION OF THE TERM POLYMORPHISM*
- *DESCRIBE THE ROLE POLYMORPHISM PLAYS IN PROGRAM DESIGN AND IMPLEMENTATION*
- *DESCRIBE WHEN COMPOSITION IS AN APPROPRIATE DESIGN MECHANISM*
- *STATE THE DEFINITION OF THE TERM POLYMORPHIC CONTAINMENT*
- *DESCRIBE WHY COMPOSITION IS CONSIDERED A FORCE MULTIPLIER*
- *UTILIZE INHERITANCE, INTERFACES, COMPOSITION, AND POLYMORPHISM TOGETHER TO ACHIEVE OPTIMAL DESIGN*

INTRODUCTION

I want to focus your attention again on the topics of inheritance, interfaces, composition, and polymorphism — the four enablers of object-oriented design and programming. I introduced you to these topics earlier in the book in their isolated contexts but now I'd like to present them to you collectively to highlight several important issues regarding their utilization in program design. At this point in the text you should be familiar with these concepts and comfortable with Java. This will be the case especially if you've attempted several of the more challenging suggested projects.

Inheritance, interfaces, composition, and polymorphism are employed together to achieve an optimal object-oriented design and implementation. However, there are no guarantees that your design, and the resulting implementation, will be anything close to optimal unless you understand the ramifications of your design decisions.

The great photographer Ansel Adams so completely mastered the photographic arts that he could produce the scene he visualized by expertly manipulating every phase of the process from exposure to print. So too must you visualize the desired characteristics of the end system and effectively employ object-oriented analysis, design, and implementation techniques to achieve your goal.

In this chapter I will review the concepts and principles of inheritance and composition. I will discuss how each contributes to code reuse and offer guidance on how to choose between the two design approaches. I will then discuss the benefits of interfaces and show you how to utilize interfaces to break functional dependencies between code modules.

As you gain programming experience and begin to grasp the subtleties and nuances of good object-oriented design you will encounter programmers and architects who have adopted one particular design methodology and extol its virtues with religious fervor. In reality, there is no absolute right object-oriented design. (*Although I do believe there are absolute wrong ones!*) A particular design's suitability is dictated primarily by application requirements. And although application requirements can be expected to evolve over time, a design, no matter how good, cannot be expected to graciously accommodate major shifts in application requirements that derive from a complete misunderstanding of the application's intended purpose.

You can, however, with the right amount of forethought, create an application architecture that accommodates major and minor feature additions with little or no negative impact to existing code modules. But you must have foreseen and accounted for the requirement to extend and change the application before making your first design decision. These issues are at the heart of the material in this chapter.

INHERITANCE VS. COMPOSITION: THE GREAT DEBATE

A continuing debate simmers amongst object-oriented design theorists and practitioners. I draw your attention to this debate only because sooner or later you will encounter it either in person or by reading the existing literature.

Each group is philosophically divided into three camps: 1) the compositionists, and 2) the inheritists, and 3) the design pragmatists. The issue revolves around the answer to this question: "*What is the preferred approach to achieving code reuse within an object-oriented program?*"

The compositionists are the most radical. They believe the only way to achieve code reuse within a program is through compositional design. All forms of inheritance as a reuse mechanism are suspect and should be avoided. They believe that anyone who advocates inheritance as a code reuse mechanism is a heretic and would be better burned at the stake than left free to wreak havoc on object-oriented programs.

The inheritists believe that inheritance is a valid form of code reuse. They know of the term composition but in their textbooks they provide the topic only superficial treatment at best. I get the distinct impression that an inheritist has little or no practical experience in object-oriented programming in a production environment.

The design pragmatists understand the meaning of the term "*engineering trade-off*". They use inheritance where it makes sense to do so and composition when the situation dictates. They understand what it means to compromise in order to make progress while at the same time taking every practical measure to ensure design goals are achieved. They realize several important facts of life: 1) the world in which we live may be perfect but human understanding of the world is decidedly imperfect, 2) it follows, then, that a human-derived model of any problem will suffer from

imperfection, 3) mapping an imperfect model to a object-oriented design results in an imperfect design, 4) mapping the imperfect design to an implementation results in an imperfect implementation, and 5) Java itself suffers from imperfection, as do all object-oriented programming languages humans create. *Therefore — to argue perfect design is a waste of time.*

Design pragmatists will attempt to achieve reuse at every step. My use of the term reuse here includes reuse in all its forms. (*code, design, knowledge, etc.*) To a design pragmatist, therefore, the issue is not strictly reuse, but rather what amount of design is appropriate to the task at hand.

WHAT'S THE END GAME?

Most programming endeavors are business ventures; there's a very real need to make a profit. If every class in every system had to be reusable in every context all projects would end in failure. So, the right amount of design depends on system requirements. Not functional requirements, per se, but implicit requirements common to all object-oriented application architectures that concern ease of maintenance, modularity, flexibility, and reusability.

Theorists can afford to argue design aesthetics all the live-long day, but production coders cannot. They must deliver the goods on what always seems to be a tight schedule. Production programmers continuously make engineering trade-offs. So long as they are not completely off target design-wise they'll make their schedule with a product that's not impossible to maintain.

I take the metaphor of "hitting the design target" literally. On a recent project I gathered my developers around a white board periodically to conduct a design gut check. I'd draw a target on the board complete with a bullseye in the center. I would then ask each developer to place a mark on the target indicating where they thought our design and coding efforts placed us, with the bullseye representing 100% perfection. We never hit the bullseye; no project ever will. What mattered most was that we did not completely miss the target, and that with each iteration we moved closer to the bullseye.

Since hitting the design bullseye on your first shot is unlikely, what then are you trying to achieve in terms of design and implementation? The answer is simply that you don't want to program yourself into a corner from which there is no escape. Your application architecture must be flexible so that it can accommodate change, it must be modular and reliable, and it must be stable.

Flexible Application Architectures

Application architectures must be flexible enough to accommodate anticipated feature additions gracefully. Graceful change accommodation is an application requirement, and in most cases this application requirement is not explicitly stated. To achieve this requirement you must have it in mind at the start of your design, otherwise you will end up eroding the application's architectural foundation as you try and shoehorn new features into the application.

Modularity And Reliability

Application components must be both modular and reliable. In an object-oriented application the natural boundary for modularity is at the class level. A class represents an abstraction of a problem domain entity. If you do a good job at maximizing class cohesion (*i.e., give the class a focused purpose*) and minimizing class coupling (*i.e., limiting its dependency on other classes*) you will find it significantly easier to reuse such classes.

Reliability does not necessarily follow from modularity, and indeed, code reliability depends upon a multitude of factors, but well-designed classes (*i.e., maximally cohesive and minimally coupled*) lend themselves to more thorough testing. A class that is designed to serve one purpose and that is reused in many locations within an application will tend to have its behavior exhaustively tested.

Architectural Stability Via Managed Dependencies

Application architectures must be stable. This means that a change to the application must have predictable results. The ability to correctly anticipate the effects that change will have on an application varies inversely with the number of inter-module dependencies. The more dependencies, the harder it is to anticipate the effects of change.

KNOWING WHEN TO ACCEPT A DESIGN THAT'S GOOD ENOUGH

The answer to the question of when a design has reached “good enough” is always the same — *it depends*. It depends on the application’s intended purpose and its associated requirements. The decision is usually made in the context of time spent making the application architecture completely generic (*never a requirement I’ve personally encountered*) vs. crafting an architecture that’s flexible enough to accommodate contextually similar feature additions.

Quick Review

There are three philosophical camps regarding the attainment of code reuse within an application: compositionists, inheritists, and design pragmatists. Design pragmatists utilize the full spectrum of object-oriented design mechanisms to achieve reuse on all possible fronts. Their approach to design is rooted in making intelligent engineering trade-offs.

The end game of good design is an application architecture that is flexible, modular, reliable, and stable.

INHERITANCE-BASED DESIGN

As you learned in chapter 11, inheritance plays a critical role in object-oriented design and implementation. However, as with all design strategies, it should be applied in right measure. In this section I want to raise your awareness of the appropriate uses of inheritance and the different forms an inheritance hierarchy can assume. Following these discussions the Person-Employee inheritance example originally presented in chapter 11 will be examined in the context of Meyer’s inheritance taxonomy and Coad’s inheritance criteria.

THREE GOOD REASONS TO USE INHERITANCE

There are at least three good reasons to use inheritance: 1) it provides you with an object-oriented design mechanism that enables you to think and reason about the structure and behavior of your code in terms of generalized and specialized classes, 2) it offers a measure of code reuse within your program, and 3) it provides you with a way to incrementally develop code.

AS A MEANS TO REASON ABOUT CODE BEHAVIOR

Thoughtfully designed inheritance hierarchies help tame conceptual complexity. If you are fortunate enough to correctly formulate the abstractions (*base classes/supertypes*) at the upper-most level of the hierarchy then you can make reasonable assumptions about the behavior of the concrete implementations (*derived classes/subtypes*) when they are used in situations expecting supertype behavior. Well-designed inheritance hierarchies enable *polymorphic behavior* which is the cornerstone of object-oriented programming.

TO GAIN A MEASURE OF CODE REUSE

Classes may contain code that can be potentially reused within your application. You need look no further than to the Java platform API for an example. The key to gaining code reuse via inheritance is to have correctly modeled the application domain in the class hierarchy in the first place and placed common behavior in classes that sit at the root of the inheritance hierarchy. (*The root in this case means the top since inheritance hierarchies are typically modeled as inverted tree structures.*)

TO FACILITATE INCREMENTAL DEVELOPMENT

Inheritance facilitates incremental development by allowing programmers to extend existing classes (*i.e. adopt existing behavior*) when necessary and appropriate. Complex applications are typically built in an iterative fashion.

Initially, an overall application architecture is laid down and one or more application features, each satisfying one, or perhaps several, outstanding requirements, are implemented with each iterative development cycle. (See chapter 3)

FORMS OF INHERITANCE: MEYER'S INHERITANCE TAXONOMY

In this book I have favored the use of four inheritance forms: *subtype*, *extension*, *functional variation*, and *implementation*. However, there are many forms of inheritance I have not discussed. These can be seen in Bertrand Meyer's Inheritance Taxonomy shown in figure 22-1. Table 22-1 provides a brief description of each inheritance form. (Note: The lightly shaded rows of table 22-1 highlight the most often used inheritance forms.) Readers interested in a complete treatment of each inheritance form are referred to Meyer's book which is listed in the references section at the end of this chapter.

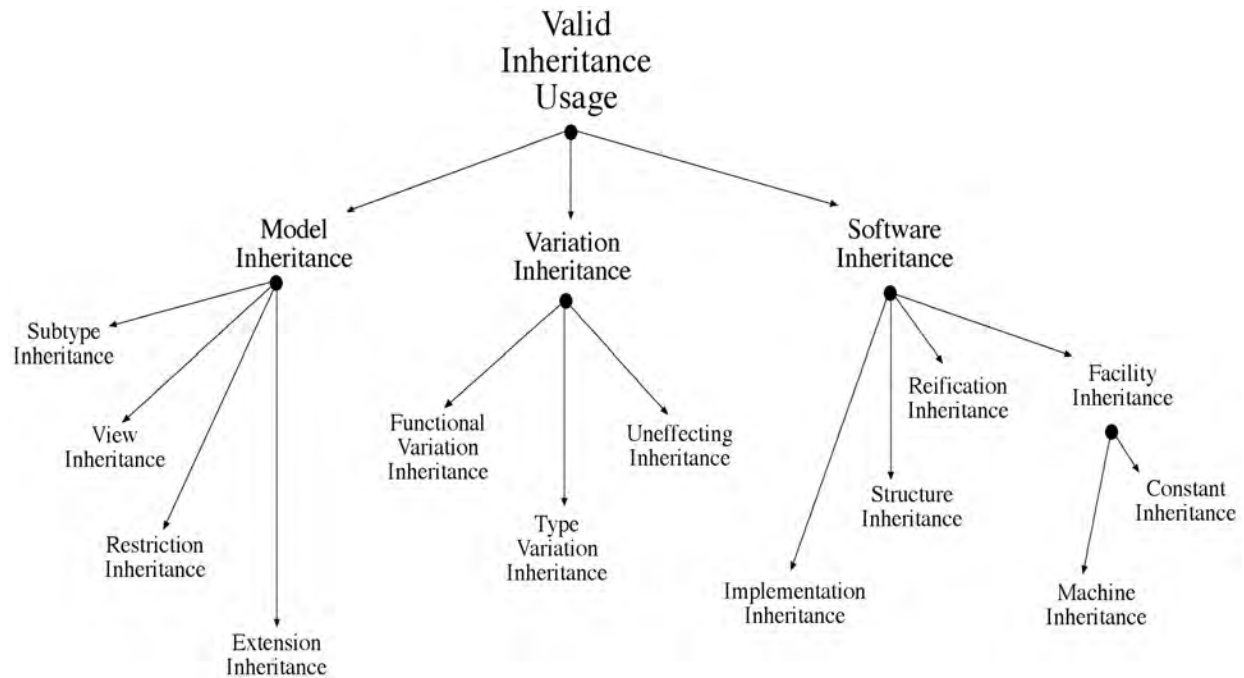


Figure 22-1: Meyer's Inheritance Taxonomy

Inheritance Form	Inheritance Form	Inheritance Form Or Description
Model	Subtype	The most obvious form of inheritance. Used to model application domain objects into categories (<i>base classes</i>) and subcategories (<i>derived classes</i>). Base classes serve to specify behavior only and are therefore abstract. (Or, in Java, an <i>interface</i> .) Derived classes represent separate and distinct types.
	Restriction	Derived classes introduce a constraint upon base class behavior. The constraint is usually applied to the base class <i>invariant</i> . (An <i>invariant</i> is a property that must hold true at all times. Invariants are discussed in detail in chapter 24.)
	Extension	Derived classes introduce new behavior not found in the base class.

Table 22-1: Inheritance Form Descriptions

Inheritance Form	Inheritance Form	Inheritance Form Or Description	
	View	Derived classes do not fit nicely into disjoint types. Subclasses do not represent distinct types but rather various ways of classifying instances of the base class. View inheritance is best applied when base and derived classes are abstract (<i>or interfaces</i>).	
Variation	Functional Variation	Derived classes redefine (<i>override</i>) base class methods.	
	Type Variation	Derived classes redefine base class method signatures. This type of inheritance is not authorized in Java. An overriding method in a derived class must have the exact method signature, including return type, of the base class method it's overriding.	
	Uneffecting Inheritance	A derived class redefines a non-abstract base class method into an abstract method. This effectively removes the unwanted base class behavior.	
Software	Reification	The base class represents a general kind of data structure, say a linked-list, and the derived class wants to adopt the functionality of the linked-list with the intent of making it into a different kind of data structure behavior-wise, say a queue. In the case of reification inheritance, the base class provides behavior (<i>non-abstract</i>).	
	Structure	Structure inheritance differs from reification inheritance in that the base class is abstract and provides only a set of specifications for the behavior of the data structure. (<i>abstract methods</i>). The derived class may provide full or partial implementation of the behavior specified by the base class. This form of inheritance occurs frequently in the Java Collections API.	
	Implementation	The derived class inherits the behavior specified by the base class and uses it as-is.	
	Facility	Constant	The base class consists of static final fields (constants) and methods whose bodies are executed only once to return a reference to a common object. A method that returned a singleton instance would fit the bill.
		Machine	The base class consists of methods the derived class finds useful to perform its mission.

Table 22-1: Inheritance Form Descriptions

COAD'S INHERITANCE CRITERIA

Peter Coad, in his book *Java Design: Building Better Apps And Applets*, provides a set of five checkpoints that can be used to ensure the effective use of inheritance. The inheritance form(s) each checkpoint seeks to avoid is listed in parentheses.

1. The derived class models an “is a special kind of,” relationship to the base class not an “is a role played by a” relationship. (*view*)
2. The derived class never needs to transmute to be an object in some other class. (*view*)
3. The derived class extends rather than overrides or nullifies the base class. (*functional, uneffecting*)
4. The baseclass is not merely a utility class representing functionality you would simply like to reuse. (*constant, machine*)

5. The inheritance hierarchy you are trying to build represents special kinds of roles, transactions, or devices within the application domain.

PERSON - EMPLOYEE EXAMPLE REVISITED

Given Meyer’s taxonomy of inheritance forms and Coad’s criteria for the effective use of inheritance, let’s reevaluate the Person-Employee inheritance hierarchy originally presented in chapter 11. It would be helpful if you print out the source code for this example to refer to while reading the assessment presented in this section. Figure 22-2 gives the UML class diagram.

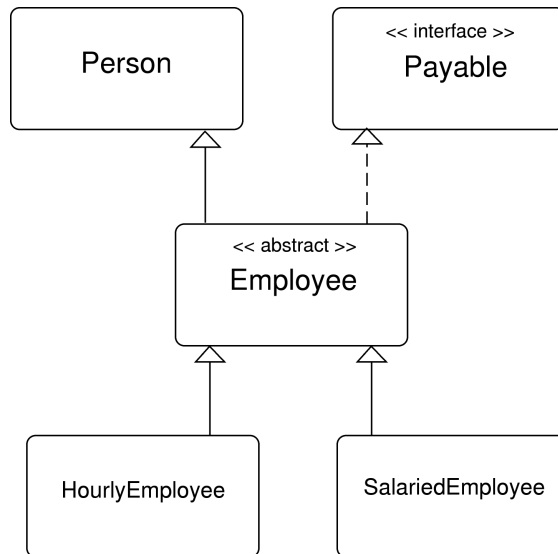


Figure 22-2: Person-Employee Inheritance Diagram

Referring to figure 22-2 — the Person class provides complete functionality for a generic person. The Person class is fully implemented and therefore not abstract. The Employee class utilizes *implementation inheritance* by extending the Person class, and *subtype inheritance* by implementing the Payable interface. However, since the Employee class fails to provide an implementation for the Payable interface’s pay() method it is declared to be abstract and pushes the responsibility of pay()’s ultimate implementation to its derived classes. The HourlyEmployee and SalariedEmployee classes both employ *implementation*, *subtype*, and *functional variation* inheritance since each fully accepts Employee’s Person-based behavior, each is a subtype of Employee, which is a subtype of both Payable and Person, and each overrides the pay() method to provide custom-derived class functionality.

When evaluated against Coad’s criteria this design fails many of the checkpoints: 1) The Employee class does not strictly model an “is a special kind of” relationship between itself and the Person base class, 2) The Person class, which sits at the root of the inheritance hierarchy, does not model a role, transaction, or device, and 3) although not evident in this limited example, subclasses may need to transmute to other subclass types. This difficulty might not be encountered until we are asked to extend the current design in response to a seemingly innocent feature request and found we had programmed ourselves into a tight corner indeed.

The following questions arise from this example: “Can this design be improved and how?” and “Is the current design completely invalid and unusable?” I will address these questions after discussing the role of interfaces, polymorphism, and compositional design in the following sections.

Quick Review

There are at least three good reasons to use inheritance: 1) it provides you with an object-oriented design mechanism that enables you to think and reason about the structure and behavior of your code in terms of generalized and specialized classes, 2) it offers a measure of code reuse within your program, and 3) it provides you with a way to incrementally develop code.

The most often used forms of inheritance include subtype, extension, functional variation, and implementation. Coad's criteria provides five checkpoints that can be used to validate the use of inheritance.

THE ROLE OF INTERFACES

The Java interface construct provides the means to specify type behavior. Interfaces can inherit from other interfaces. Interfaces can also include constant definitions. This facilitates the application of a rich variety of inheritance forms to include: *subtype inheritance*, *extension inheritance*, and *constant inheritance*.

A Java class can inherit from only one other class (*by using the extends keyword*), but can implement any number of interfaces. This is one of the key advantages of using interfaces; any class, from any inheritance hierarchy, can implement any interface as required. Such classes are not tied to the static typing enforced by their inheritance hierarchy; they can become any type they need to be by simply implementing the required interface. For example, you can make any class an ActionListener by implementing the ActionListener interface and providing behavior for its actionPerformed() method.

REDUCING OR LIMITING INTERMODULE DEPENDENCIES

Another powerful advantage interfaces provide is the ability to reduce intermodule dependencies upon concrete implementation classes. If you program to an interface you free the code from its dependency on any particular implementation of that interface. (*The same effect can be achieved by programming to abstract classes.*) Any object that implements the interface can be used where objects of that interface type are called for in the code. (*polymorphic substitution*)

However, simply using interfaces does not automatically result in reduced functional dependencies. You must also be aware that you cannot fully eliminate all functional dependencies from your code, but you can architecturally organize your application in a way that limits them to a handful of classes. It's kind of like corralling cattle; the beasts must be rounded up and concentrated.

One way to approach such a task is to use the Factory class pattern. A Factory is an object that is used to create instances of objects of a specified type. This type is usually an interface. (*Either an interface proper or an abstract class.*) In practice, there are usually two interfaces involved in employing the Factory class pattern; 1) the interface that corresponds to the type of objects the factory creates, and 2) an interface to the Factory itself so that when the application starts up different factory instances can be used.

The Factory class pattern is usually employed together with the Singleton pattern since only one instance of a factory object is utilized by all objects that need objects of the type the factory creates. Factory and Singleton patterns are covered formally in chapter 25.

MODELING DOMINANT, COLLATERAL, AND DYNAMIC ROLES

The obvious question arises: "*When should you model an application domain entity or concept as an interface, a class, or as a hierarchy of interfaces or classes?*"

Interfaces serve primarily as behavioral specifications and thus they are the natural choice for implementing subtype inheritance hierarchies. (*subtype inheritance*) And since one interface can extend another existing interface, and in the process specify additional behavior (*i.e., add more method declarations specific to the subtype*), they can also be used to implement extension inheritance. But classes can be used to implement the same types of inheritance forms, so how do you choose between the two?

I would suggest that when modeling dominant roles favor the use of a class hierarchy, and when modeling collateral roles favor the use of interfaces. When attempting to model the many possible roles some object might assume dynamically during application runtime, model these as a collection of Strings (*i.e., a Vector of Strings perhaps*), contained within the object. These concepts are discussed in detail below.

Dominant Roles

A *dominant role* is one that is unlikely to change or transmute once modeled. An example of this would be the Employee - HourlyEmployee or Employee - SalariedEmployee inheritance relationships. Behavior can be safely implemented in the base class and subclasses can extend base class functionality as required. However, you must always keep in mind how you intend to polymorphically utilize objects within the application. Such a consideration might mean the difference between choosing *functional variation inheritance* over *extension inheritance*. This topic will be explored more thoroughly later in the Applied Polymorphism section.

The use of interfaces is not precluded when modeling dominant roles and in some cases an interface will serve as the root type of a dominant role class hierarchy. An example of this is offered later in the chapter.

Collateral Roles

A *collateral role* is one that is likely to be utilized in combination with other dominant or collateral roles but, once modeled, is unlikely to change. An example of a collateral role can be found in the ActionListener interface, which is frequently implemented along with other interfaces such as MouseMotionListener, WindowListener, etc. So, for example, if you need a new class that is a JDialog you extend JDialog. (*JDialog is a dominant role.*) If this same class also needs to be an ActionListener and MouseMotionListener (*collateral roles*) then you need to implement these interfaces as well.

Dynamic Roles

In Java, an object's type cannot be dynamically changed at runtime. Therefore dominant and collateral roles, once modeled, are static in nature. However, an object often needs to assume roles dynamically. An example of when this is necessary can be found in applications where a user has access to different levels of application functionality based upon the type of access authority they have been granted. (*The "user" application domain entity might be represented by a class named "User" .*) These types of user access roles are dynamic because a user might be granted increased or decreased access rights at application runtime. Role types such as these are often stored in persistent storage (*i.e., relational databases*). An application that utilizes such roles to restrict user access is usually modeled as an Access Control Graph (ACG).

Quick Review

The Java interface construct provides the means to specify type behavior and supports a rich variety of inheritance forms to include: *subtype inheritance*, *extension inheritance*, and *constant inheritance*. Interfaces, when used in conjunction with the Factory and Singleton patterns, can reduce inter-module functional dependencies to a handful of classes.

When modeling dominant roles favor the use of a class hierarchy. When modeling collateral roles favor the use of interfaces. When attempting to model the many possible roles some object might assume dynamically during application runtime, model these as a collection of Strings contained within the object.

Applied Polymorphism

Chapter 11 touched on the topic of polymorphic behavior and conceptually it is easy to grasp. Polymorphism is the ability to treat different objects in the same manner. In object oriented programming this means that your program utilizes references to base class types (*preferably interfaces or abstract class types*) that, at runtime, actually contain references to derived class objects.

You must plan for the proper use of polymorphic behavior from the moment you start laying the foundation of your application architecture. This means you must consider carefully your choice of inheritance forms when designing your class inheritance hierarchies.

Since the goal of polymorphic programming is the uniform treatment of derived class objects it follows that derived classes should conform to the interface specified by the base class. It also follows, then, that the preferred

form of inheritance to satisfy this requirement would be functional variation, where the base class specifies behavior via an abstract method and derived classes override the method to provide a custom implementation.

Using extension inheritance instead of functional variation adds complications. For example, if a derived class extends the behavior of a base class by defining additional methods, then objects of this new type, accessed via a base class reference, must be cast to the proper type before the new functionality can be accessed.

Refer to the Person-Employee class diagram shown in figure 22-2. There are four ways to get polymorphic behavior from this inheritance hierarchy: 1) create an Object type reference and initialize it to point to either an HourlyEmployee or SalariedEmployee type object, 2) create a Person type reference and initialize it to point to either an HourlyEmployee or SalariedEmployee type object, 3) create an Employee type reference and initialize it to point to either an HourlyEmployee or SalariedEmployee type object, or 4) create a Payable type reference and initialize it to point to either an HourlyEmployee or SalariedEmployee type object. Each approach places restrictions on what functionality can be accessed via the reference without casting.

The Object reference will only allow methods defined in the Object class to be called. If a Person reference is used then Person and Object methods can be called. If a Payable reference is used then only the pay() method can be called, while the use of the Employee reference allows Person, Object, and Payable methods to be called.

Quick Review

Polymorphism is the ability to treat different objects in the same manner. In object oriented programming this means that your program utilizes references to base class types (*preferably interfaces or abstract class types*) that, at runtime, actually contain references to derived class objects.

You must plan for the proper use of polymorphic behavior from the moment you start laying the foundation of your application architecture. This means you must consider carefully your choice of inheritance forms when designing your class inheritance hierarchies.

Composition-Based Design As A Force Multiplier

Good compositional design has its foundations in the thorough understanding of inheritance, interfaces, and polymorphism. Compositional design acts as a force multiplier in that it combines the power of all these design techniques. In this regard you should never be forced to choose composition over inheritance, but rather, you should apply composition in a way that compliments inheritance, considers the use of interfaces, and keeps the goal of polymorphic behavior in mind from the very beginning.

Two Types Of Aggregation

An object is an aggregate if it contains and uses the services of other objects. An aggregate object consists of itself (*the whole*) and the objects it contains (*its parts*).

Recall from chapter 10 that there are two types of aggregation: 1) simple aggregation, and 2) composite aggregation. Simple aggregation occurs when the whole object does not control the lifetime of its part objects. Conversely, a composite aggregate has complete control over the lifetime of its part objects.

Polymorphic Containment

An aggregate object is dependent upon the behavior of its part objects. Complex aggregates may comprise many different types of part objects, each providing specialized behavior. A careful consideration of polymorphic behavior can offer a uniform treatment of these differing part types. This can be achieved via polymorphic containment where the whole class targets the interface(s) of its part class(es), treats its part objects as a collection of parts, and obtains its part objects from a part object factory.

In some cases polymorphic containment may not be strictly necessary. An example of this would be when the concrete part class is considered fairly stable design-wise, meaning there is a low probability that its interface will change during the application's maintenance lifetime. This is an example of an engineering trade-off.

AN EXTENDED EXAMPLE

Considering all that's been said about inheritance, interfaces, polymorphic behavior, and compositional design, I'd like to present a different approach to the Person-Employee example. Figure 22-3 gives the class diagram for the complete application.

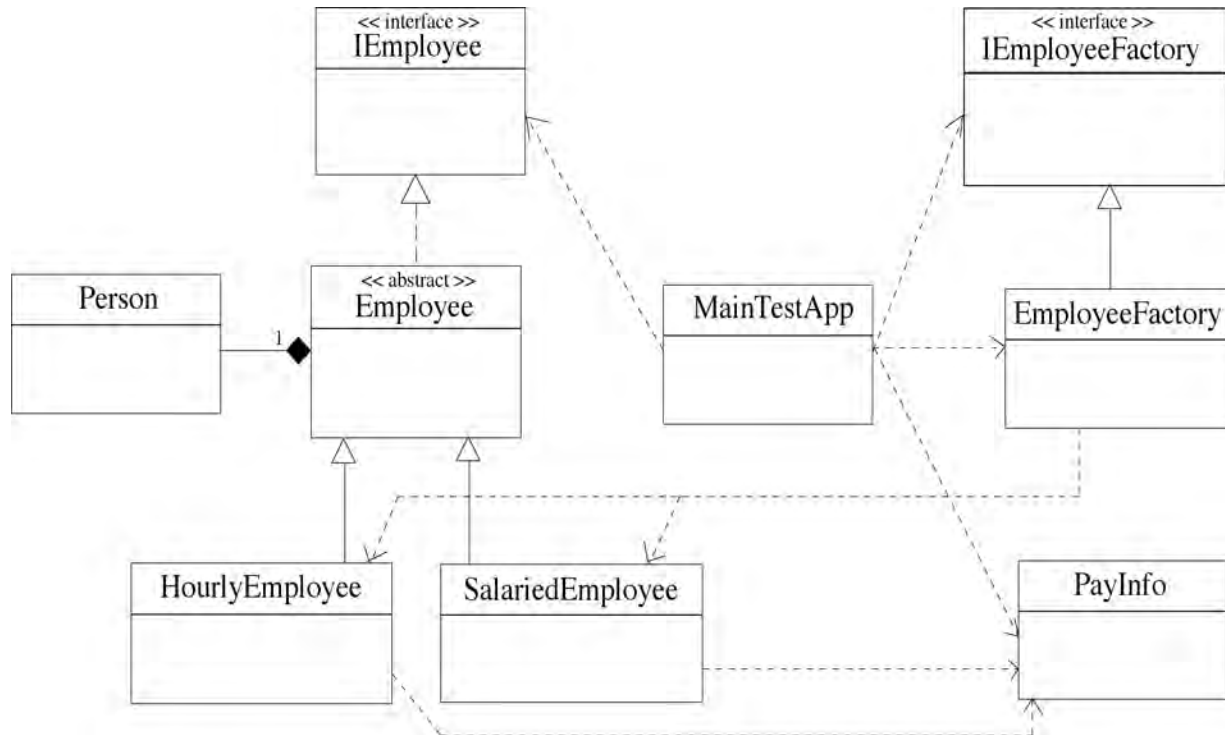


Figure 22-3: Revised Person - Employee Example

Referring to figure 22-3 — the `IEmployee` interface serves as the interface specification for employee objects. The `Employee` class no longer inherits from `Person`. Instead, the `Employee` class implements the `IEmployee` interface and contains a `Person` object by value. The rationale for this design decision might be as follows: a company has a number of employee positions available. These positions are filled by people. In most corporations, there is a permanent association between an employee position, signified by an employee number, and the person who fills the position. (*If you leave the company and return you are usually assigned the same employee number.*)

Having the `Employee` class contain the `Person` class breaks the inheritance relationship between `Person` and `Employee`, and, as you learned earlier, based on Coad's Criteria, this inheritance relationship was invalid from the start. The revised `Employee - HourlyEmployee` and `Employee - SalariedEmployee` inheritance hierarchy models dominant roles within the company (*and within the app*) that are unlikely to change. For instance, there will always be a clear distinction between hourly employees and salaried employees.

The `MainTestApp` class has dependencies upon the `IEmployee` interface, the `IEmployeeFactory` interface, the `EmployeeFactory` class, and the `PayInfo` class. The `EmployeeFactory` has a dependency upon the `HourlyEmployee` and `SalariedEmployee` classes, but limits this dependency to their constructors. The `HourlyEmployee` and `SalariedEmployee` classes each depend upon the `PayInfo` class. However, this dependency is not necessarily bad in that it is unlikely that the `PayInfo` class will undergo future change.

There is a simple association between the `IEmployeeFactory` interface and the `IEmployee` interface as well as between the `PayInfo` class, `IEmployee`, and the classes in the `Employee` inheritance hierarchy, but they have been omitted from the diagram for clarity. Examples 22.1 through 22.9 provide the code for this application.

22.1 `IEmployee.java`

```

1      public interface IEmployee {
2          int getAge();
3          String getFullName();
4          String getNameAndAge();
5          String getFirstName();
6          String getMiddleName();

```



```

7     String getLastName();
8     String getGender();
9     String getEmployeeNumber();
10    void setBirthday(int year, int month, int day);
11    void setFirstName(String f_name);
12    void setMiddleName(String m_name);
13    void setLastName(String l_name);
14    void setGender(String gender);
15    void setEmployeeNumber(String emp_no);
16    void setPayInfo(PayInfo pi);
17    double getPay();
18    String toString();
19 }

```

22.2 *Employee.java*

```

1     public abstract class Employee implements IEmployee {
2         private Person _person = null;
3         private String _employee_number = null;
4
5         protected Employee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
6             int dob_day, String gender, String employee_number){
7             _person = new Person(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender);
8             _employee_number = employee_number;
9         } // end constructor
10
11        public int getAge() { return _person.getAge(); }
12        public String getFullName() { return _person.getFullName(); }
13        public String getNameAndAge() { return _person.getNameAndAge(); }
14        public String getFirstName() { return _person.getFirstName(); }
15        public String getMiddleName() { return _person.getMiddleName(); }
16        public String getLastName() { return _person.getLastName(); }
17        public String getGender() { return _person.getGender(); }
18        public String getEmployeeNumber() { return _employee_number; }
19        public void setBirthday(int year, int month, int day) { _person.setBirthday(year, month, day); }
20        public void setFirstName(String f_name) { _person.setFirstName(f_name); }
21        public void setMiddleName(String m_name) { _person.setMiddleName(m_name); }
22        public void setLastName(String l_name) { _person.setLastName(l_name); }
23        public void setGender(String gender) { _person.setGender(gender); }
24        public void setEmployeeNumber(String emp_no) { _employee_number = emp_no; }
25        public String toString(){ return _person.getNameAndAge() + " " + _employee_number; }
26        public abstract void setPayInfo(PayInfo pi); // defer implementation
27        public abstract double getPay(); // defer implementation
28
29    } // end Employee class definition

```

22.3 *Person.java*

```

1     import java.util.*;
2
3     public class Person {
4         private String first_name = null;
5         private String middle_name = null;
6         private String last_name = null;
7         private Calendar birthday = null;
8         private String gender = null;
9
10        public static final String MALE = "Male";
11        public static final String FEMALE = "Female";
12
13        public Person(String f_name, String m_name, String l_name, int dob_year,
14            int dob_month, int dob_day, String gender){
15            first_name = f_name;
16            middle_name = m_name;
17            last_name = l_name;
18            this.gender = gender;
19
20            birthday = Calendar.getInstance();
21            birthday.set(dob_year, dob_month, dob_day);
22        }
23
24        public int getAge(){
25            Calendar today = Calendar.getInstance();
26            int now = today.get(Calendar.YEAR);
27            int then = birthday.get(Calendar.YEAR);
28            return (now - then);
29        }
30
31        public String getFullName(){ return (first_name + " " + middle_name + " " + last_name); }
32
33        public String getFirstName(){ return first_name; }

```

```

34     public void setFirstName(String f_name) { first_name = f_name; }
35
36     public String getMiddleName(){ return middle_name; }
37     public void setMiddleName(String m_name){ middle_name = m_name; }
38
39     public String getLastName(){ return last_name; }
40     public void setLastName(String l_name){ last_name = l_name; }
41
42     public String getNameAndAge(){ return (getFullName() + " " + getAge()); }
43
44     public String getGender(){ return gender; }
45     public void setGender(String gender){ this.gender = gender; }
46
47     public void setBirthday(int year, int month, int day){ birthday.set(year, month, day); }
48
49 } //end Person class definition

```

22.4 HourlyEmployee.java

```

1     public class HourlyEmployee extends Employee {
2         private double _hours_worked = 0;
3         private double _hourly_rate = 0.0;
4
5         public HourlyEmployee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
6             int dob_day, String gender, String employee_number){
7             super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender, employee_number);
8         }
9
10        public void setPayInfo(PayInfo pi){
11            _hours_worked = pi.getHoursWorked();
12            _hourly_rate = pi.getHourlyRate();
13        }
14
15        public double getPay() { return _hours_worked * _hourly_rate; }
16
17        public String toString() { return super.toString() + " $" + getPay(); }
18
19    } // end HourlyEmployee class definition

```

22.5 SalariedEmployee.java

```

1     public class SalariedEmployee extends Employee {
2         private double _salary = 0;
3
4         public SalariedEmployee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
5             int dob_day, String gender, String employee_number){
6             super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender, employee_number);
7         }
8
9         public void setPayInfo(PayInfo pi){
10            _salary = pi.getSalary();
11        }
12
13        public double getPay() { return (_salary/12.0)/2.0; }
14
15        public String toString() { return super.toString() + " $" + getPay(); }
16    } // end SalariedEmployee class definition

```

22.6 PayInfo.java

```

1     public class PayInfo {
2         private double _salary = 0;
3         private double _hours_worked = 0;
4         private double _hourly_rate = 0;
5
6         public PayInfo(double salary){
7             _salary = salary;
8         }
9
10        public PayInfo(double hours_worked, double hourly_rate){
11            _hours_worked = hours_worked;
12            _hourly_rate = hourly_rate;
13        }
14
15        public PayInfo(){ }
16
17        public double getHoursWorked(){ return _hours_worked; }
18        public double getHourlyRate(){ return _hourly_rate; }
19        public double getSalary(){ return _salary; }
20
21    } // end PayInfo class definition

```

22.7 *IEmployeeFactory.java*

```

1  public interface IEmployeeFactory {
2      IEmployee getNewSalariedEmployee(String f_name, String m_name, String l_name, int dob_year,
3          int dob_month, int dob_day, String gender, String employee_number);
4      IEmployee getNewHourlyEmployee(String f_name, String m_name, String l_name, int dob_year,
5          int dob_month, int dob_day, String gender, String employee_number);
6  } // end IEmployeeFactory interface definition

```

22.8 *EmployeeFactory.java*

```

1  public class EmployeeFactory implements IEmployeeFactory {
2
3
4      public IEmployee getNewSalariedEmployee(String f_name, String m_name, String l_name, int dob_year,
5          int dob_month, int dob_day, String gender,
6          String employee_number){
7          return new SalariedEmployee(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender,
8              employee_number);
9      }
10
11     public IEmployee getNewHourlyEmployee(String f_name, String m_name, String l_name, int dob_year,
12         int dob_month, int dob_day, String gender,
13         String employee_number){
14         return new HourlyEmployee(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender,
15             employee_number);
16     }
17
18
19
20 } // end EmployeeFactory class definition

```

22.9 *MainTestApp.java*

```

1  public class MainTestApp {
2
3      private IEmployeeFactory _employee_factory = null;
4      private IEmployee[] _employee_array = null;
5
6
7      public MainTestApp(){
8          _employee_factory = new EmployeeFactory();
9          _employee_array = new IEmployee[2];
10
11     }
12
13     public void createEmployees(){
14         _employee_array[0] = _employee_factory.getNewSalariedEmployee("Rick", "W", "Miller", 1977, 03,
15             04, "Male", "12345");
16         _employee_array[0].setPayInfo(new PayInfo(123000.00));
17         _employee_array[1] = _employee_factory.getNewHourlyEmployee("Laura", "Jean", "Richter", 1980, 05,
18             06, "Female", "16273");
19         _employee_array[1].setPayInfo(new PayInfo(40, 45.00));
20     }
21
22     public void showEmployeeInformation(){
23         for(int i = 0; i < _employee_array.length; i++){
24             System.out.println(_employee_array[i]);
25         }
26     }
27
28     public static void main(String[] args){
29         MainTestApp mta = new MainTestApp();
30         mta.createEmployees();
31         mta.showEmployeeInformation();
32     }
33
34 } // end MainTestApp class definition

```

Referring to example 22.9 — the `MainTestApp` has two private fields: one of type `IEmployeeFactory` and the other an array of type `IEmployee`. The constructor method on line 7 initializes the `_employee_factory` reference to point to an `EmployeeFactory` instance. The constructor also initializes the `_employee_array` reference, giving it a length of two.

The `createEmployees()` method beginning on line 13 uses the `_employee_factory` reference to create an `HourlyEmployee` and a `SalariedEmployee`, assigning the reference to each `IEmployee` object to an element of the `_employee_array`. (*Remember, the `EmployeeFactory` returns references to objects that implement the `IEmployee`*

interface.) The `createEmployees()` method then sets each employee's pay information by calling the `setPayInfo()` method with a new `PayInfo` object as an argument.

The `showEmployeeInformation()` method beginning on line 22 simply steps through the `_employee_array` and prints information about each employee to the console.

The `main()` method starts on line 28 and creates an instance of the `MainTestApp` class followed by a call to the `createEmployees()` and `showEmployeeInformation()` methods respectively. Figure 22-4 shows the results of running this program.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/person_emp_mod_2] swadog% java MainTestApp
Rick W Miller 28 12345 $5125.0
Laura Jean Richter 25 16273 $1800.0
[Rick-Millers-Computer:~/desktop/person_emp_mod_2] swadog%
```

Figure 22-4: Results of Running Example 22.9

Quick Review

Good compositional design has its foundations in the thorough understanding of inheritance, interfaces, and polymorphism. Compositional design acts as a force multiplier in that it combines the power of all these design techniques. In this regard you should never be forced to choose composition over inheritance, but rather, you should apply composition in a way that compliments inheritance, considers the use of interfaces, and keeps the goal of polymorphic behavior in mind from the very beginning.

An object is an aggregate if it contains and uses the services of other objects. An aggregate object consists of itself (*the whole*) and the objects it contains (*its parts*).

There are two types of aggregation: 1) simple aggregation, and 2) composite aggregation. Simple aggregation occurs when the whole object does not control the lifetime of its part objects. Conversely, a composite aggregate has complete control over the lifetime of its part objects.

Complex aggregates may comprise many different types of part objects, each providing specialized behavior. A careful consideration of polymorphic behavior can offer a uniform treatment of these differing part types. This can be achieved via polymorphic containment where the whole class targets the interface(s) of its part class(es), treats its part objects as a collection of parts, and obtains its part objects from a part object factory.

SUMMARY

There are three philosophical camps regarding the attainment of code reuse within an application: compositionists, inheritists, and design pragmatists. Design pragmatists utilize the full spectrum of object-oriented design mechanisms to achieve reuse on all possible fronts. Their approach to design is rooted in making intelligent engineering trade-offs.

The end game of good design is an application architecture that is flexible, modular, reliable, and stable.

There are at least three good reasons to use inheritance: 1) it provides you with an object-oriented design mechanism that enables you to think and reason about the structure and behavior of your code in terms of generalized and specialized classes, 2) it offers a measure of code reuse within your program, and 3) it provides you with a way to incrementally develop code.

The most often used forms of inheritance include subtype, extension, functional variation, and implementation. Coad's criteria provides five checkpoints that can be used to validate the use of inheritance.

The Java interface construct provides the means to specify type behavior and supports a rich variety of inheritance forms to include: *subtype inheritance*, *extension inheritance*, and *constant inheritance*. Interfaces, when used in conjunction with the Factory and Singleton patterns, can reduce inter-module functional dependencies to a handful of classes.

When modeling dominant roles favor the use of a class hierarchy. When modeling collateral roles favor the use of interfaces. When attempting to model the many possible roles some object might assume dynamically during application runtime, model these as a collection of Strings contained within the object.

Polymorphism is the ability to treat different objects in the same manner. In object-oriented programming this means that your program utilizes references to base class types (*preferably interfaces or abstract class types*) that, at runtime, actually contain references to derived class objects.

You must plan for the proper use of polymorphic behavior from the moment you start laying the foundation of your application architecture. This means you must consider carefully your choice of inheritance forms when designing your class inheritance hierarchies.

Good compositional design has its foundations in the thorough understanding of inheritance, interfaces, and polymorphism. Compositional design acts as a force multiplier in that it combines the power of all these design techniques. In this regard you should never be forced to choose composition over inheritance, but rather, you should apply composition in a way that compliments inheritance, considers the use of interfaces, and keeps the goal of polymorphic behavior in mind from the very beginning.

An object is an aggregate if it contains and uses the services of other objects. An aggregate object consists of itself (*the whole*) and the objects it contains (*its parts*).

There are two types of aggregation: 1) simple aggregation, and 2) composite aggregation. Simple aggregation occurs when the whole object does not control the lifetime of its part objects. Conversely, a composite aggregate has complete control over the lifetime of its part objects.

Complex aggregates may comprise many different types of part objects, each providing specialized behavior. A careful consideration of polymorphic behavior can offer a uniform treatment of these differing part types. This can be achieved via polymorphic containment where the whole class targets the interface(s) of its part class(es), treats its part objects as a collection of parts, and obtains its part objects from a part object factory.

Skill-Building Exercises

1. **Further Research:** Obtain Meyer's book (*listed in the references section*) and read the chapters related to inheritance.
2. **Further Research:** Obtain Coad's book (*listed in the references section*) and read the section that talks about the five inheritance checkpoints. (*Coad's Criteria*)
3. **Further Research:** Obtain Martin's book (*listed in the references section*) and read the chapter on designing the employee payroll system.
4. **Programming:** Compile and execute the example code listed in this chapter.
5. **UML Drill:** Create a UML sequence diagram of the main() method of the MainTestApp class given in example 22.9.
6. **Applied Object-Oriented Theory:** Evaluate the Aircraft Engine Simulation code given in chapter 11 from the standpoint of Meyer's Inheritance Taxonomy and Coad's Criteria.
7. **Further Research:** Explore the topic of Access Control Graphs.
8. **Applied Polymorphism:** Consider the following interface and class definitions then answer the following questions:

```
1     public interface IFoo {
2         void a();
3         void b();
4     }
```

```

1     public class Bar implements IFoo {
2         public void a(){ }
3         public void b(){ }
4         public void c(){ }
5         public void d(){ }
6     }

```

- a. What methods can be called without casting if a reference of type IFoo is declared and initialized to point to an object of type Bar?
- b. What types of inheritance forms are applied in this example?

9. Identify Inheritance Form: Consider the following code:

```

1     public class BaseClass {
2         public void f(){ System.out.println("Hello from BaseClass f()!"); }
3     }

1     public class DerivedClass extends BaseClass {
2         public void f() { System.out.println("Hello from DerivedClass f()!"); }
3     }

```

What type of inheritance form is applied in this example?

10. Identify Inheritance Form: If a functional linked-list is extended to create a new type of data structure, what type of inheritance form is being applied?

SUGGESTED PROJECTS

1. **RobotRat Encore Une Fois:** Revisit the RobotRat project presented in chapter 3. Use the object-oriented approach to redesign the application so that different types of remote controlled objects can be moved around the floor. Consider the rat's pen, and the concept of its position upon the floor as separate entities that comprise a Rat. Give a remote controlled object the capability to display itself as a graphic or text representation. You may implement this project as a stand-alone application or as a multi-threaded client-server application or applet.
2. **Networked Home Appliance Control System:** Design and implement a networked home appliance control system. Assume all appliances have a unique IP address. When an appliance is connected to the network it automatically registers with the central controller. The following systems are connected to the appliance network: house climate control sub-system (*includes automatic windows, air conditioning and heating*), hot water heater, lights, oven and refrigerator.
3. **Hi-Tech Building Security System:** Design and implement a building security system that includes different types of identity verification sensors to include voice recognition, finger print recognition, retina scan recognition, key-pad and card-swipe entry.
5. **Biological And Radiological Hazard Detection System:** Your country needs your object-oriented design and programming talents! Design and implement a biological and radiological hazard detection system for the cities of the world. Various sensor types will be utilized to detect nuclear radiation (*alpha, beta, and gamma particles*), and different types of poisonous gases to include sarin, chlorine, and mustard. Your system must be able to monitor sensor status which includes the sensor's geographic location.

SELF-TEST QUESTIONS

1. What are the three essential purposes of inheritance?
2. What is meant by the term *engineering trade-off*?
3. List at least three benefits provided by inheritance.
4. What's the purpose of an interface?
5. What's the difference between an interface and an abstract class?
6. Why is compositional design considered to be a force multiplier?
7. What is meant by the term *polymorphism*?
8. How much design is good enough?
9. What is the fundamental unit of modularity in an object-oriented program?
10. What are the five checkpoints of Coad's Criteria?

REFERENCES

Bertrand Meyer. *Object-Oriented Software Construction*, Second Edition. Prentice Hall PTR, Upper Saddle River, New Jersey. ISBN: 0-13-629155-4

Grady Booch. *Object-Oriented Analysis And Design With Applications*, Second Edition. The Benjamin/Cummings Publishing Company, Inc. Redwood City, CA. ISBN: 0-8053-5340-2

Robert C. Martin. *Designing Object-Oriented C++ Applications Using The Booch Method*. Prentice Hall, Englewood Cliffs, New Jersey. ISBN: 0-13-203837-4

Peter Coad, et. al. *Java Design: Building Better Apps And Applets*, Second Edition. Prentice Hall PTR, Upper Saddle River, New Jersey. ISBN: 0-13-911181-6

James Gosling, et. al. *The Java™ Language Specification*, Second Edition. Addison-Wesley, Boston, MA. ISBN: 0-201-31008-2

Antero Taivalsaari. *On the Notion of Inheritance*. ACM Computing Surveys, Vol. 28, No. 3, September 1996.

Barbara Liskov, John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Boston, MA. ISBN: 0-201-65768-6

Various Contributors. *Inheritance(object-oriented programming)*. From Wikipedia, the free encyclopedia. [[http://en.wikipedia.org/wiki/Inheritance_\(object-oriented_programming\)](http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))]

Rick Miller. *C++ For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-02-8

NOTES

CHAPTER 23



TWO CHILDREN

WELL-BEHAVED OBJECTS

LEARNING OBJECTIVES

- STATE THE IMPORTANCE OF WELL-BEHAVED JAVA OBJECTS
- LIST AND DESCRIBE THE REQUIRED FUNCTIONALITY OF A WELL-BEHAVED JAVA OBJECT
- STATE THE PURPOSE OF THE `hashCode()` METHOD
- DESCRIBE TWO ALGORITHMS FOR IMPLEMENTING A SUITABLE `hashCode()` METHOD
- DEMONSTRATE YOUR ABILITY TO IMPLEMENT A SUITABLE `hashCode()` METHOD
- DESCRIBE HOW JAVA COLLECTIONS FRAMEWORK CLASSES UTILIZE THE `hashCode()` METHOD
- DESCRIBE HOW JAVA COLLECTIONS FRAMEWORK CLASSES UTILIZE THE `equals()` METHOD
- STATE THE REASONS FOR OVERRIDING THE `toString()` METHOD
- STATE THE PURPOSE OF THE `equals()` METHOD
- DESCRIBE THE CONTRACT FOR IMPLEMENTING THE `equals()` METHOD
- DESCRIBE THE PROPER USE OF THE `clone()` METHOD
- EXPLAIN THE DIFFERENCE BETWEEN A DEEP COPY VS. A SHALLOW COPY
- DESCRIBE THE PROPER USE OF THE `finalize()` METHOD
- LIST AND DESCRIBE THE DIFFERENCES BETWEEN THE `Comparator` AND `Comparable` INTERFACES
- DESCRIBE WHAT IT MEANS FOR A CLASS OF OBJECTS TO HAVE A NATURAL ORDERING
- STATE THE PURPOSE OF THE `Serializable` INTERFACE

INTRODUCTION

It is imperative that you fully understand and take steps to ensure the proper behavior of user-defined class objects within a program. Objects are considered to be well-behaved when they behave as expected and pull no surprises.

The types of behavior I refer to deal primarily with seven usage scenarios: 1) when two objects of the same type are compared against each other for equality, 2) when an object is cloned, 3) when an object is inserted into a Hash-table collection, 4) when an object needs to provide a string representation of itself, 5) when two objects of the same type must be compared to each other to determine their natural ordering, 6) when two objects of the same type must be compared to each other to determine their total ordering, and 7) when an object is serialized.

The purpose of this chapter then is to show you how to create classes whose objects behave as expected in these seven usage scenarios. Here you will learn how to override the Object class's `toString()`, `hashCode()`, `equals()`, and `clone()` methods along with why, and under what conditions, these methods should be overridden. I will show you how to implement the Comparable and Comparator interfaces and then show you how to use objects of these types in a collection. I will also review the need for a class to implement the Serializable interface if you intend to save objects to disk or send them over a network.

The skills you learn here will save you hours of wasted time and frustration chasing down subtle, hard to trace bugs that result from ill-behaved objects.

CONSIDER OBJECT USAGE FROM THE START

The need for well-behaved objects must be considered during the application design phase. All classes should be evaluated against the seven usage scenarios listed in table 23-1.

Yes/No	Usage Scenario	Implementation Requirement
	Will objects of this type be required to provide a string representation of themselves?	Override the <code>Object.toString()</code> method. (<i>Although not a strict requirement, it is a generally accepted practice to override the <code>toString()</code> method in all cases.</i>)
	Will objects of this type be compared against each other for equality?	Override <code>Object.equals()</code> method obeying the general contract specified in the Java API documentation. Also override the <code>Object.hashCode()</code> method.
	Will objects of this type be inserted into a hash-based collection?	Override <code>Object.hashCode()</code> method obeying the general contract specified in the Java API documentation. Also override the <code>Object.equals()</code> method.
	Will objects of this type be cloned?	Implement the <code>java.lang.Cloneable</code> interface and override the <code>Object.clone()</code> method. If the object being cloned contains other objects or collections of objects be sure to perform a deep copy.
	Do objects of this type have a natural ordering?	Implement the <code>java.lang.Comparable</code> interface.
	Will objects of this type be subjected to a total ordering?	Create a Comparator class by implementing the <code>java.util.Comparator</code> interface. This class should also implement the <code>java.io.Serializable</code> interface if it will be used to compare serializable data structures.
	Will objects of this type be saved to disk or sent via a network?	Implement the <code>java.io.Serializable</code> interface.

Table 23-1: Object Usage Scenario Evaluation Checklist

Referring to table 23-1 — not all usage scenarios will apply to all classes. However, it is generally considered good Java programming practice for all classes to override the `Object.toString()` method as it comes in handy during debugging.

Also notice that the `Object.equals()` and `Object.hashCode()` methods are usually overridden together. This is because the correct behavior of one has implications for the correct behavior of the other. These two methods also bring up the notion of a behavior contract. A contract is a specification of expected method behavior. If you override a method but fail to honor the contract then your objects will behave erratically.

Method behavior contracts are found in the Java API documentation. The general contract specifications for the `equals()` and `hashCode()` methods are found in the documentation for the `java.lang.Object` class.

Applying THE OBJECT USAGE SCENARIO CHECKLIST

Let's apply the object usage scenario checklist to our old friend the `Person` class originally presented in chapter 11 and recently used again in chapter 22.

Yes/No	Usage Scenario	Implementation Requirement
YES	Will objects of this type be required to provide a string representation of themselves?	Override the <code>Object.toString()</code> method. (<i>Although not a strict requirement, it is a generally accepted practice to override the <code>toString()</code> method in all cases.</i>) - <i>Override <code>toString()</code> in <code>Person</code>.</i>
YES	Will objects of this type be compared against each other for equality?	Override <code>Object.equals()</code> method obeying the general contract specified in the Java API documentation. Also override the <code>Object.hashCode()</code> method. - <i>Override <code>equals()</code> in <code>Person</code>.</i>
YES	Will objects of this type be inserted into a hash-based collection?	Override <code>Object.hashCode()</code> method obeying the general contract specified in the Java API documentation. Also override the <code>Object.equals()</code> method. - <i>Override <code>hashCode()</code> in <code>Person</code>.</i>
YES	Will objects of this type be cloned?	Implement the <code>java.lang.Cloneable</code> interface and override the <code>Object.clone()</code> method. If the object being cloned contains other objects or collections of objects be sure to perform a deep copy. - <i>Cloning means making an independent copy of an object. There may be times when we need to ensure that <code>Person</code> objects can be properly cloned, for instance, when a collection contains references to <code>People</code> objects. If such a collection is cloned, all the sub-objects contained in the collection must be cloneable as well. The <code>Person</code> class will implement the <code>Cloneable</code> interface and override the <code>clone()</code> method.</i>
YES	Do objects of this type have a natural ordering?	Implement the <code>java.lang.Comparable</code> interface. - <i><code>Person</code> will implement <code>Comparable</code>.</i>
YES	Will objects of this type be subjected to a total ordering?	Create a <code>Comparator</code> class by implementing the <code>java.util.Comparator</code> interface. This class should also implement the <code>java.io.Serializable</code> interface if it will be used to compare serializable data structures. - <i>A new <code>Comparator</code> class will be created to give a total ordering to <code>Person</code> objects.</i>

Table 23-2: Applying The Object Usage Scenario Evaluation Checklist

Yes/No	Usage Scenario	Implementation Requirement
YES	Will objects of this type be saved to disk or sent via a network?	Implement the java.io.Serializable interface. - <i>Person will implement Serializable.</i>

Table 23-2: Applying The Object Usage Scenario Evaluation Checklist

Throughout the rest of this chapter the Person class will be modified to behave well in the usage scenarios indicated above. Example 23.1 gives the listing for the Person class that will be used as the baseline for these performance enhancements.

23.1 Person.java (Baseline version)

```

1      import java.util.*;
2
3      public class Person {
4          private String first_name = null;
5          private String middle_name = null;
6          private String last_name = null;
7          private Calendar birthday = null;
8          private String gender = null;
9
10         public static final String MALE = "Male";
11         public static final String FEMALE = "Female";
12
13         public Person(String f_name, String m_name, String l_name, int dob_year, int dob_month,
14             int dob_day, String gender){
15             first_name = f_name;
16             middle_name = m_name;
17             last_name = l_name;
18             this.gender = gender;
19
20             birthday = Calendar.getInstance();
21             birthday.set(dob_year, dob_month, dob_day);
22         }
23
24         public int getAge(){
25             Calendar today = Calendar.getInstance();
26             int now = today.get(Calendar.YEAR);
27             int then = birthday.get(Calendar.YEAR);
28             return (now - then);
29         }
30
31         public String getFullName(){ return (first_name + " " + middle_name + " " + last_name); }
32
33         public String getFirstName(){ return first_name; }
34         public void setFirstName(String f_name) { first_name = f_name; }
35
36         public String getMiddleName(){ return middle_name; }
37         public void setMiddleName(String m_name){ middle_name = m_name; }
38
39         public String getLastName(){ return last_name; }
40         public void setLastName(String l_name){ last_name = l_name; }
41
42         public String getNameAndAge(){ return (getFullName() + " " + getAge()); }
43
44         public String getGender(){ return gender; }
45         public void setGender(String gender){ this.gender = gender; }
46
47         public void setBirthday(int year, int month, int day){ birthday.set(year, month, day); }
48
49     } //end Person class

```

Quick Review

Anticipated object usage scenarios must be considered during the application design phase. All classes should be evaluated against the seven object usage scenarios. Use the Object Usage Scenario Evaluation Checklist given in table 23-1 to help evaluate your class design.

OVERRIDING JAVA.LANG.OBJECT METHODS

The Object class sits at the root of all Java reference type classes. The majority of its methods are meant to be overridden in derived classes. This is especially true if objects of the derived class type will be used in the scenarios listed in table 23-1. The Object methods you should know how to properly override include toString(), equals(), hashCode(), clone(), and finalize(). I discuss these methods in detail below.

OVERRIDING toString()

The toString() method returns a string representation of the object in question. It is generally considered to be good programming practice to override the toString() method in all classes. Doing so can significantly aid debugging efforts. Detailed object information can help you pinpoint problem objects more quickly than if you relied on the default toString() method behavior provided by the Object class.

What, exactly, constitutes a satisfactory toString() method for a particular class is completely subjective. A good toString() method for the Person class might be one that returns a string representation of all its fields as is shown in example 23.2.

23.2 Person.toString()

```

1     public String toString(){
2         return this.getFullName() + " " + gender + " " + birthday.get(Calendar.DATE) + "/"
3             + birthday.get(Calendar.MONTH) + "/" + birthday.get(Calendar.YEAR);
4     }

```

In this example, the birthday field is used to extract the day (DATE), month, and year of the person's birth. This information is concatenated with the string provided by the Person.getFullName() method.

OVERRIDING equals() AND hashCode()

The equals() and hashCode() methods are usually overridden together since the behavior of one has implications for the behavior of the other.

equals() Method

The equals() method is overridden when objects of a particular class support the notion of being logically equal to each other. The equals() method must satisfy a set of five behaviors (*equivalence relation*) as specified in the Java API documentation. Table 23-3 lists the equals() method's equivalence relation behaviors.

The equals() method must be...	Description
Reflexive	For any non-null reference value x, x.equals(x) should return true.
Symmetric	For any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
Transitive	For any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
Consistent	For any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
...and lastly...	For any non-null reference value x, x.equals(null) should return false.

Table 23-3: equals() Method Equivalence Relation

Overriding the equals() method and getting the equivalence relation right requires some thought and work. The best place to start is to determine whether or not the equals() method must be overridden in the first place. Joshua Bloch, in his book *Effective Java™ Programming Language Guide*, suggests applying four criterion to make this determination. If any of these criterion apply you may not need to override the equals() method. Bloch's equals() Method Criterion are listed in table 23-3.

Criterion Applies? Yes/No	Criterion
	Each instance of the class is inherently unique.
	You don't care if the class provides a logical equality test.
	The superclass already overrides equals() and the inherited behavior is appropriate for this class.
	The class is private or package-private and you are reasonably sure that its equals() method will never be invoked.

Table 23-4: Bloch's equals() Method Criteria

The Person class fails Bloch's test on all four counts. Each instance of Person is not inherently unique, I care that it provides a logical equality test, and although its direct base class (*java.lang.Object*) does provide a default implementation for equals(), its default behavior is not appropriate for my purposes. Lastly, it's a public class and the equals() method will most likely be invoked at some point in the future. Therefore it's a good idea to override equals() in Person and ensure the correct behavior.

OVERRIDING EQUALS() IN THE PERSON CLASS

What does it mean for one instance of Person to be logically equal to another instance? The meaning of logical equality is entirely at the mercy of design. Is it enough to simply compare birthdays, names, and genders?

Consider also for a moment whether or not the Person class contains enough attributes to make valid logical comparisons. Although in real life each person is unique, in an application, an instance of a Person object is made logically unique by the values of its attributes. (*That is, two distinct Person objects might have identical names, birthdays, and genders. These two objects would be logically equal if only their names, birthdays, and genders were used to make the comparison, but would this scratch the itch?*) While you chew on that for a while I will proceed to override the equals() method in the Person class using the existing Person class attributes. Example 23.3 gives the code for the Person class's equals() method.

23.3 Person.equals()

```

1      public boolean equals(Object o){
2          if(o == null) return false;
3          boolean is_equal = false;
4          if(o instanceof Person){
5              if(this.first_name.equals(((Person)o).first_name) &&
6                  this.middle_name.equals(((Person)o).middle_name) &&
7                  this.last_name.equals(((Person)o).last_name) && this.gender.equals(((Person)o).gender) &&
8                  (this.birthday.get(Calendar.YEAR) == ((Person)o).birthday.get(Calendar.YEAR)) &&
9                  (this.birthday.get(Calendar.MONTH) == ((Person)o).birthday.get(Calendar.MONTH)) &&
10                 (this.birthday.get(Calendar.DATE) == ((Person)o).birthday.get(Calendar.DATE)) ){
11                  is_equal = true;
12              }
13          }
14          return is_equal;
15      }

```

Referring to example 23.3 — the Person.equals() method returns false immediately if the incoming reference is null. It then initializes the boolean return variable is_equal to false and then checks to see if the incoming reference is an instance of the Person class. If not the comparison is complete and the method returns false. Otherwise, the field-by-field comparison between objects takes place in the body of the if statement beginning on line 5. If everything checks out the is_equal variable is set to true and the method returns.

TESTING PERSON.TOSTRING() AND PERSON.EQUALS() METHODS

Now is a good time to test both the toString() and equals() methods. Example 23.4 gives the code for a short test driver program named MainTestApp that creates a few Person objects, calls the toString() method on each, and then runs the equals() method through a complete test of its equivalence relation requirements.

23.4 MainTestApp.java

```

1      public class MainTestApp {
2          public static void main(String[] args){
3              Person p1 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
4              Person p2 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
5              Person p3 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
6              Person p4 = new Person("Steve", "J", "Jones", 1932, 9, 20, Person.MALE);
7
8              System.out.println("P1: " + p1.toString());
9              System.out.println("P2: " + p2.toString());
10             System.out.println("P3: " + p3.toString());
11             System.out.println("P4: " + p4.toString());
12
13             System.out.println(p1.equals(p1)); // reflexive - should be true
14             System.out.println(p1.equals(p2) && p2.equals(p1)); // symmetric - should be true
15             System.out.println(p1.equals(p2) && p2.equals(p3) && p1.equals(p3)); // transitive - should be true
16             System.out.println(p1.equals(p2)); // consistent - should be true every time this app executes
17             System.out.println(p1.equals(p4)); // consistent - should be false every time this app executes
18             System.out.println(p1.equals(null)); // should always return false
19         }
20     } // end MainTestApp

```

Referring to example 23.4 — four Person objects are created and assigned to references p1 through p4. References p1 through p3 are logically equivalent because they are initialized with identical information. (*Although they are distinct objects within the Java runtime environment.*) Reference p4 is logically different from the rest as you can see. Lines 8 through 11 test the toString() method on each object. Lines 13 through 18 run through the equivalence relation test for reflexivity, symmetry, etc. Figure 23-1 shows the results of running this program.

```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/overriding_object_methods] swodog% java MainTestApp
P1: Rick W Miller Male 6/5/1963
P2: Rick W Miller Male 6/5/1963
P3: Rick W Miller Male 6/5/1963
P4: Steve J Jones Male 20/9/1932
true
true
true
true
false
false
[Rick-Millers-Computer:~/desktop/overriding_object_methods] swodog%

```

Figure 23-1: Results of Running Example 23.4

hashCode() METHOD

Now that the Person.equals() method works properly attention can be turned to the implementation of its hashCode() method.

The hashCode() method returns an integer which is referred to as the object's hash value. The default implementation of hashCode() found in the Object class will, in most cases, return a unique hash value for each distinct object even if they are logically equivalent. In most cases this default behavior will be acceptable, however, if you intend to use a class of objects as keys to hashtables or other hash-based data structures, then you must override the hashCode() method and obey the general contract as specified in the Java API documentation. The general contract for the hashCode() is given in table 23-5.

Check	Criterion
	The hashCode() method must consistently return the same integer when invoked on the same object more than once during an execution of a Java application provided no information used in equals() comparisons on the object is modified. This integer need not remain constant from one execution of an application to another execution of the same application.
	The hashCode() method must produce the same results when called on two objects if they are equal according to the equals() method.
	The hashCode() method is not required to return distinct integer results for logically unequal objects, however, failure to do so may result in degraded hashtable performance.

Table 23-5: The hashCode() General Contract

As you can see from table 23-5 there is a close relationship between the equals() and hashCode() methods. It is recommended that any fields used in the equals() method comparison be used to calculate an object's hash code. Remember, the primary goal when implementing a hashCode() method is to have it return the same value consistently for logically equal objects. It would also be nice if the hashCode() method returned distinct hash code values for logically unequal objects but according to the general contract this is not a strict requirement.

Before actually implementing the Person.hashCode() method I want to provide you with two hash code generation algorithms.

BLOCH'S HASH CODE GENERATION ALGORITHM

Joshua Bloch, in his book *Effective Java™ Programming Language Guide*, provides the following algorithm for calculating a hash code:

1. Start by storing a constant, nonzero value in an int variable called result. (*Josh used the value 17*)
2. For each significant field *f* in your object (*each field involved in the equals() comparison*) do the following:
 - a. Compute an int hash code *c* for the field:
 - i. If the field is boolean compute: $(f ? 0 : 1)$
 - ii. If the field is a byte, char, short, or int, compute: $(int) f$
 - iii. If the field is a long compute: $(int) (f \gg 32)$
 - iv. If the field is a float compute: `Float.floatToIntBits(f)`
 - v. If the field is a double compute: `Double.doubleToLongBits(f)`, and then hash the resulting long according to step 2.a.iii.
 - vi. If the field is an object reference and this class's equals() method compares the field by recursively invoking equals(), recursively invoke hashCode() on the field. If a more complex comparison is required, compute a "canonical representation" for this field and invoke hashCode() on the canonical representation. If the value of the field is null, return 0.
 - vii. If the field is an array, treat it as if each element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine these values in step 2.b
 - b. Combine the hash code *c* computed in step a into result as follows:

```
result = 37*result + c;
```
3. Return result.
4. If equal object instances do not have equal hash codes fix the problem!

ASHMORE'S HASH CODE GENERATION ALGORITHM

Derek Ashmore, in his book *The J2EE Architect's Handbook: How To Be A Successful Technical Architect For J2EE Applications*, recommends the following simplified hash code algorithm:

1. Concatenate the required fields (*those involved in the equals() comparison*) into a string.

2. Call the hashCode() method on that string.
3. Return the resulting hash code value.

Overriding hashCode() In The Person Class

For our purposes, Ashmore's algorithm will work fine. Since the Person.toString() method concatenates all the same fields together that are used in the Person.equals() method, it should return the same hash code for logically equal object instances and different hash codes for logically unequal objects. Example 23.5 gives the code for the Person.hashCode() method.

23.5 Person.hashCode()

```
1     public int hashCode(){
2         return this.toString().hashCode();
3     }
```

Referring to example 23.5 — it's short, sweet, and to the point. But will it work as expected?

Testing Person.hashCode()

Example 23.6 gives the code for the modified MainTestApp program that tests the hashCode() method on four Person objects. Figure 23-2 shows the results of running this program.

23.6 MainTestApp.java (mod 1)

```
1     public class MainTestApp {
2         public static void main(String[] args){
3             Person p1 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
4             Person p2 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
5             Person p3 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
6             Person p4 = new Person("Steve", "J", "Jones", 1932, 9, 20, Person.MALE);
7
8             System.out.println("P1: " + p1.toString());
9             System.out.println("P2: " + p2.toString());
10            System.out.println("P3: " + p3.toString());
11            System.out.println("P4: " + p4.toString());
12
13            System.out.println(p1.equals(p1)); // reflexive - should be true
14            System.out.println(p1.equals(p2) && p2.equals(p1)); // symmetric - should be true
15            System.out.println(p1.equals(p2) && p2.equals(p3) && p1.equals(p3)); // transitive - should be true
16            System.out.println(p1.equals(p2)); // consistent - should be true every time this app executes
17            System.out.println(p1.equals(p4)); // consistent - should be false every time this app executes
18            System.out.println(p1.equals(null)); // should always return false
19
20            System.out.println();
21            System.out.println(p1.hashCode());
22            System.out.println(p2.hashCode());
23            System.out.println(p3.hashCode());
24            System.out.println(p4.hashCode());
25        }
26    } // end MainTestApp
```

```
Terminal - tcsh
[Rick-Millers-Computer:~/desktop/overriding_object_methods] swoog$ java MainTestApp
P1: Rick W Miller Male 6/5/1963
P2: Rick W Miller Male 6/5/1963
P3: Rick W Miller Male 6/5/1963
P4: Steve J Jones Male 20/9/1932
true
true
true
true
true
false
false
-1164897410
-1164897410
-1164897410
954968999
[Rick-Millers-Computer:~/desktop/overriding_object_methods] swoog$
```

Figure 23-2: Results of Running Example 23.6

Referring to example 23.6 — the `hashCode()` method is called on each `Person` reference `p1` through `p4` on lines 21 through 24. As you can see from figure 23-2, the hash codes for `Person` references `p1` through `p3` are the same and the hash code for `Person` reference `p4` is different from the rest.

OVERRIDING `clone()`

Overriding the `clone()` method is really a two-part process: 1) you must implement the `java.lang.Cloneable` interface, and 2) override the `Object.clone()` method. The `Cloneable` interface has no methods; it simply serves as a type tag to indicate to `clone()` methods whether or not this class of objects is or is not cloneable. A class is cloneable only if it implements the `Cloneable` interface. If a class does not implement the `Cloneable` interface and the `clone()` method is called on one of its objects, the `clone()` method will throw a `CloneNotSupportedException`.

Shallow Copy vs. Deep Copy

Before you go cloning objects you must be aware of the potential dangers associated with cloning. Specifically, you must understand the difference between a shallow copy vs. a deep copy. A shallow copy is one that simply copies the contents of the cloned object's reference fields into the reference fields of the new object. This results in the new object's fields referring to the same objects referred to by the cloned object's reference fields. Figure 23-3 illustrates the concept of the shallow copy.

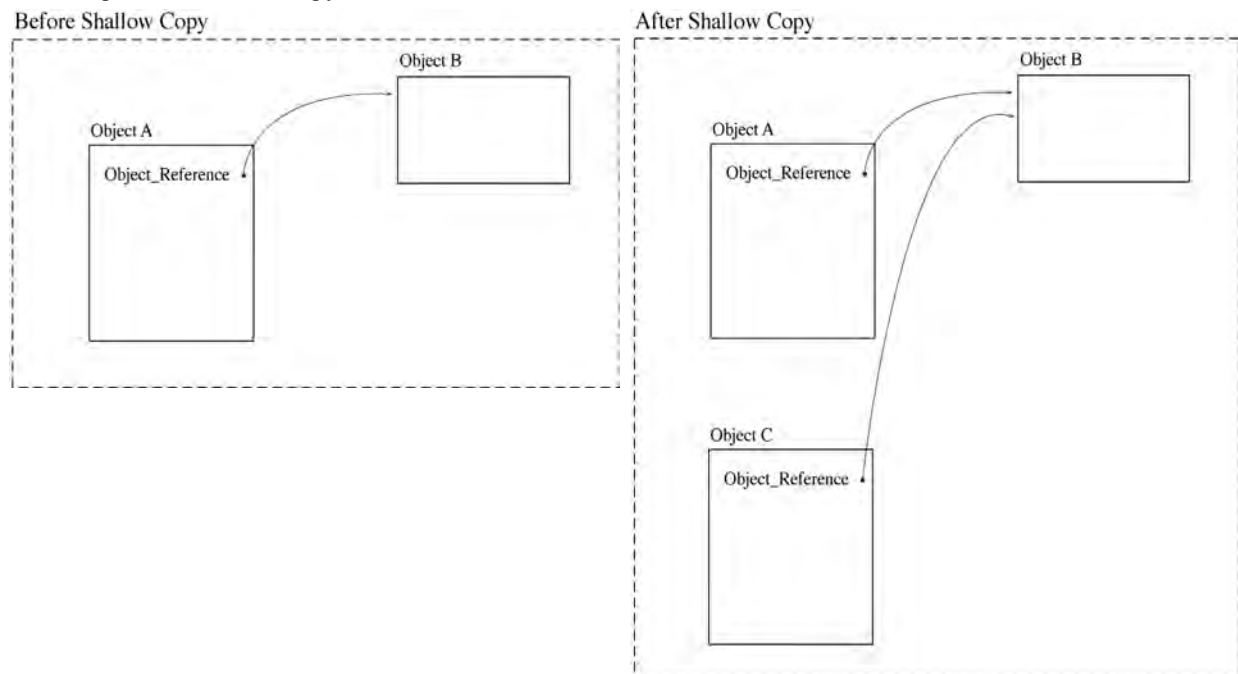


Figure 23-3: Concept of a Shallow Copy

In special cases you may want objects to share their contained instances (*containment by reference*) but doing so should be the intentional result of your application design, not because of a naive implementation of the `clone()` method.

A deep copy is different from a shallow copy in that it creates copies of the cloned object's instance field objects and assigns them to the new object. Figure 23-4 illustrates the concept of a deep copy. Referring to figure 23-4 — before `Object A` is cloned, its `Object_Reference` field points to `Object B`. A deep copy of `Object A` will result in a new object, `Object C`, whose `Object_Reference` field points to its very own copy of `Object B`, which, in this example, becomes a new object named `Object D`.

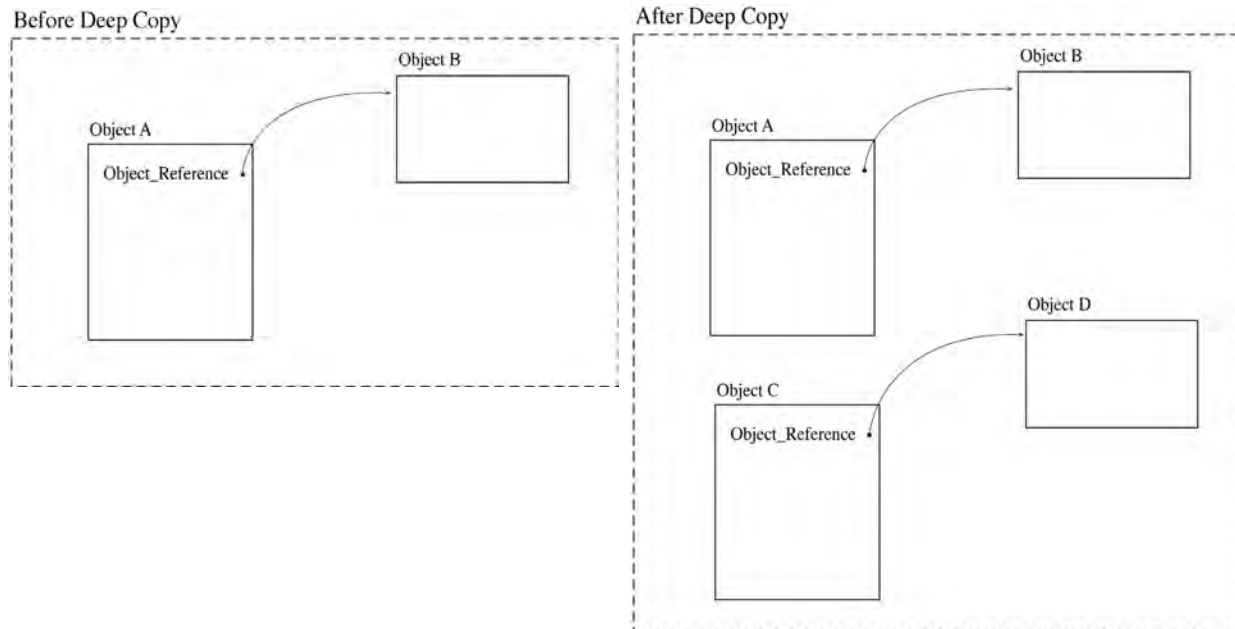


Figure 23-4: Concept of a Deep Copy

OVERRIDING PERSON.CLONE()

Example 23.7 gives the code the `Person.clone()` method. Since I am not showing the code for the entire `Person` class just note that for this method to work the `Person` class must implement the `Cloneable` interface.

```

1      public Object clone() throws CloneNotSupportedException {
2          super.clone();
3          return new Person(new String(first_name), new String(middle_name), new String(last_name),
4                          birthday.get(Calendar.YEAR), birthday.get(Calendar.MONTH),
5                          birthday.get(Calendar.DATE), new String(gender));
6      }

```

23.7 *Person.clone()*

Referring to example 23.7 — the `Person.clone()` method creates and returns a new, logically equal copy of the existing person object. Before doing so, however, it calls the base class `clone()` method. Example 23.8 gives the code for the modified version of `MainTestApp` that runs the `Person.clone()` method through its paces. Figure 23-5 gives the results of running this program.

```

1      public class MainTestApp {
2          public static void main(String[] args){
3              Person p1 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
4              Person p2 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
5              Person p3 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
6              Person p4 = new Person("Steve", "J", "Jones", 1932, 9, 20, Person.MALE);
7
8              System.out.println("P1: " + p1.toString());
9              System.out.println("P2: " + p2.toString());
10             System.out.println("P3: " + p3.toString());
11             System.out.println("P4: " + p4.toString());
12
13             System.out.println(p1.equals(p1)); // reflexive - should be true
14             System.out.println(p1.equals(p2) && p2.equals(p1)); // symmetric - should be true
15             System.out.println(p1.equals(p2) && p2.equals(p3) && p1.equals(p3)); // transitive - should be true
16             System.out.println(p1.equals(p2)); // consistent - should be true every time this app executes
17             System.out.println(p1.equals(p4)); // consistent - should be false every time this app executes
18             System.out.println(p1.equals(null)); // should always return false
19
20             System.out.println();
21             System.out.println(p1.hashCode());
22             System.out.println(p2.hashCode());
23             System.out.println(p3.hashCode());
24             System.out.println(p4.hashCode());

```

23.8 *MainTestApp.java (mod 2)*

```

25     System.out.println();
26
27     try{
28         Person p5 = (Person)p4.clone();
29         System.out.println(p4.toString());
30         System.out.println(p5.toString());
31         System.out.println(p4 == p5); // is it the same object - should be false
32         System.out.println(p4.equals(p5)); // are they logically equal - should be true
33         System.out.println(p4.hashCode());
34         System.out.println(p5.hashCode()); // hash codes should be the same
35     }catch(CloneNotSupportedException ignored){ }
36
37     }
38 } // end MainTestApp

```

```

Terminal -- tcsh
[rick-millers-computer:~/desktop/overriding_object_methods] swadog$ java MainTestApp
P1: Rick W Miller Male 6/5/1963
P2: Rick W Miller Male 6/5/1963
P3: Rick W Miller Male 6/5/1963
P4: Steve J Jones Male 28/9/1932
true
true
true
true
true
false
-1164897410
-1164897410
-1164897410
954968999
Steve J Jones Male 28/9/1932
Steve J Jones Male 28/9/1932
false
true
954968999
954968999
[rick-millers-computer:~/desktop/overriding_object_methods] swadog$

```

Figure 23-5: Results of Running Example 23.8

OVERRIDING finalize() – AND AN ALTERNATIVE APPROACH

In this section I just want to reemphasize the purpose of a `finalize()` method, focus your attention on its limitations, and suggest an alternative approach to the execution of time-sensitive object clean-up operations.

An object's `finalize()` method is called by the Java Virtual Machine garbage collector when the object is garbage collected. When an object is released from memory it should also release any system resources it may have reserved. However, there are no guarantees exactly when the `finalize()` method will get called, only that it will be called eventually. (*You saw an example of this behavior in the networked version of RobotRat presented in chapter 20.*) Therefore, it is imperative that you do not rely on a `finalize()` method to tidy up things in a time-sensitive manner.

There is no reason for the `Person` class to override the `finalize()` method. However, if you are creating a class that must release resources in a time-sensitive manner then your best bet is to implement an explicit clean-up method. Examples of this type of a method can be found throughout the Java Platform API. (*For example, to clean-up a `Socket` and release its resources you call the `Socket.close()` method.*)

Quick Review

Most of the methods provided by the `java.lang.Object` class are meant to be overridden. These include the `toString()`, `equals()`, `hashCode()`, `clone()`, and `finalize()` methods.

The purpose of the `toString()` method is to provide a string representation for a class of objects. The contents of this string are dictated by class design requirements. It's considered good programming practice to always override the `toString()` method.

The purpose of the `equals()` method is to perform a logical equality comparison between two objects of the same class. The `equals()` method implements an equivalence relation specified in the Java Platform API documentation that

says the method should be reflexive, symmetric, transitive, consistent, and should return false if a non-null instance object is compared against a null reference. The behavior of the equals() method has ramifications for the behavior of other methods as well.

The purpose of the hashCode() method is to provide an integer value for an object when it is used with hashtable or hash-based data structures. The hashCode() method has a general behavior contract specified in the Java Platform API documentation and its performance is dependent upon the behavior of the equals() method. Namely, fields utilized in the equals() method to perform the logical comparison should be used in the hashCode() method to generate the object's hash code.

The purpose of the clone() method is to create a logically equal copy of an existing object. The important thing to consider when implementing the clone() method is the undesirable effects of performing a shallow copy of complex class types.

The purpose of the finalize() method is to tie-up loose ends when an object is collected by the Java Virtual Machine (JVM) garbage collector. However, do not depend on finalize() to perform a time-sensitive operation as there are no guarantees when the finalize() method will be called.

IMPLEMENTING THE java.lang.COMPARABLE INTERFACE

If objects of a particular class are to be inserted into a sorted collection then there must be a way for the collection insertion mechanism to determine if one object is greater-than, less-than, or equal to another object of the same type. To participate in such a comparison an object must implement the Comparable interface. The Comparable interface has one method: compareTo(), which must be implemented to communicate this natural ordering.

An class's natural ordering is completely at the mercy of the design. Sometimes, a class's natural ordering is immediately evident. Take the Integer class for example. Given the values 1, 3, and 5, it's easy to see their natural ordering. But what, exactly, would constitute a natural ordering for the Person class? If we select the calculated attribute age then Person objects inserted into a collection will be sorted according to how old they are. (*either ascending or descending*) If we order them according to their last_name then they will be inserted into the collection alphabetically by last name.

The one rule that the Java API documentation recommends you follow when implementing the Comparable.compareTo() method is that it should behave consistently with the equals() method. (*It is a recommendation — not a requirement.*) This means that if there are two logically equal Person references p1 and p2, then (p1.compareTo(p2) == 0) has the same value as p1.equals(p2).

Let's see how this might be done in the Person class. Example 23.9 gives the code for the Person.compareTo() method.

```

1         public int compareTo(Object o){
2             return this.toString().compareTo(o.toString());
3         }

```

23.9 Person.compareTo()

This was too easy! As it turns out, the String class implements Comparable. Since Person overrides toString() it can be used to get string representations of each Person object. But since the Person.toString() method includes both name and birthday information, does the ordering make sense? One thing is certain, this implementation behaves consistently with equals()!

Example 23.10 gives the code for a modified version of MainTestApp that tests the Person.compareTo() method. Figure 23-9 shows the results of running this program.

```

1         public class MainTestApp {
2             public static void main(String[] args){
3                 Person p1 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
4                 Person p2 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
5                 Person p3 = new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE);
6                 Person p4 = new Person("Steve", "J", "Jones", 1932, 9, 20, Person.MALE);
7
8                 System.out.println("P1: " + p1.toString());
9                 System.out.println("P2: " + p2.toString());
10                System.out.println("P3: " + p3.toString());
11                System.out.println("P4: " + p4.toString());
12            }

```

23.10 MainTestApp.java (mod 3)

```

13     System.out.println(p1.equals(p1)); // reflexive - should be true
14     System.out.println(p1.equals(p2) && p2.equals(p1)); // symmetric - should be true
15     System.out.println(p1.equals(p2) && p2.equals(p3) && p1.equals(p3)); // transitive - should be true
16     System.out.println(p1.equals(p2)); // consistent - should be true every time this app executes
17     System.out.println(p1.equals(p4)); // consistent - should be false every time this app executes
18     System.out.println(p1.equals(null)); // should always return false
19
20     System.out.println();
21     System.out.println(p1.hashCode());
22     System.out.println(p2.hashCode());
23     System.out.println(p3.hashCode());
24     System.out.println(p4.hashCode());
25     System.out.println();
26
27     try{
28         Person p5 = (Person)p4.clone();
29         System.out.println(p4.toString());
30         System.out.println(p5.toString());
31         System.out.println(p4 == p5); // is it the same object - should be false
32         System.out.println(p4.equals(p5)); // are they logically equal - should be true
33         System.out.println(p4.hashCode());
34         System.out.println(p5.hashCode()); // hash codes should be the same
35     }catch(CloneNotSupportedException ignored){ }
36
37     System.out.println();
38     System.out.println(p1.compareTo(p2));
39     System.out.println(p1.compareTo(p4));
40     System.out.println(p4.compareTo(p1));
41
42 }
43 } // end MainTestApp

```

```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/overriding_object_methods] swodog% java MainTestApp
P1: Rick W Miller Male 6/5/1963
P2: Rick W Miller Male 6/5/1963
P3: Rick W Miller Male 6/5/1963
P4: Steve J Jones Male 20/9/1932
true
true
true
true
false
false
-1164897410
-1164897410
-1164897410
954968999
Steve J Jones Male 20/9/1932
Steve J Jones Male 20/9/1932
false
true
954968999
954968999
0
-1
1
[Rick-Millers-Computer:~/desktop/overriding_object_methods] swodog%

```

Figure 23-6: Results of Running Example 23.10

Referring to figure 23-6 — the results of executing the `compareTo()` method appear at the bottom left of the screen capture. Zero means the objects compared are equal. A negative integer, -1, means the left-hand side object (*referenced by p1*) is less than the right-hand side object. (*The object the left-hand side object is being compared against, which is p4 in this case.*) A positive integer, 1, means the left-hand side object is greater than the right-hand side object.

Quick Review

Implement the `Comparable` interface when a class of objects will be utilized in a collection and will therefore be subject to a natural ordering. It is recommended that the `Comparable.compareTo()` method perform consistently with the `equals()` method.

USING PERSON OBJECTS IN A COLLECTION

Calling the `compareTo()` method on an object explicitly is something you won't do except perhaps during testing, as was done here. Where it really comes into play is when using collections. The following example program creates several `Person` objects, inserts them into a list, then sorts the list using the `Collections.sort()` method. The results of running this program are shown in figure 23-7.

23.11 *CollectionTestApp.java*

```

1      import java.util.*;
2
3      public class CollectionTestApp {
4          public static void main(String[] args){
5              List list = new ArrayList();
6
7              list.add(new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE));
8              list.add(new Person("Debbie", "S", "Sears", 1986, 8, 3, Person.FEMALE));
9              list.add(new Person("John", "L", "Thompson", 1978, 1, 4, Person.MALE));
10             list.add(new Person("Riddic", "H", "Bean", 2001, 11, 15, Person.MALE));
11             list.add(new Person("Gloria", "J", "Albright", 1922, 12, 2, Person.FEMALE));
12             list.add(new Person("Zena", "P", "Warrior", 1968, 1, 26, Person.FEMALE));
13             list.add(new Person("Shelly", "H", "Marshall", 1993, 7, 15, Person.FEMALE));
14             list.add(new Person("Jessica", "A", "Simpson", 1912, 6, 3, Person.FEMALE));
15             list.add(new Person("Peter", "R", "Rabbit", 1999, 4, 30, Person.MALE));
16             list.add(new Person("Mohamad", "A", "Abbas", 1961, 5, 29, Person.MALE));
17             list.add(new Person("Sapna", "P", "Gupta", 1988, 8, 11, Person.FEMALE));
18             list.add(new Person("Marvin", "C", "Williams", 1945, 2, 18, Person.MALE));
19             list.add(new Person("Kyle", "V", "Miller", 1954, 9, 13, Person.MALE));
20             list.add(new Person("Joseph", "L", "Smith", 1963, 10, 23, Person.MALE));
21             list.add(new Person("Nora", "G", "Jones", 1977, 11, 4, Person.FEMALE));
22             list.add(new Person("Hedy", "E", "Lamarr", 1914, 2, 28, Person.FEMALE));
23
24             for(Iterator i = list.iterator(); i.hasNext();){
25                 System.out.println(i.next().toString());
26             }
27
28             Collections.sort(list);
29
30             System.out.println("-----Sorted List-----");
31             for(Iterator i = list.iterator(); i.hasNext();){
32                 System.out.println(i.next().toString());
33             }
34         }
35     } // end CollectionTestApp

```

Referring to figure 23-7 — The `Person.compareTo()` method causes `Person` objects to be sorted alphabetically by their string values. And since the `Person.toString()` method builds the string starting with first name, it appears at first that this is how they are being sorted.

IMPLEMENTING JAVA.UUTIL.COMPARATOR INTERFACE

The `Comparator` interface is meant to be implemented by objects that will be used to impose a total ordering upon other objects. A total ordering is an ordering imposed upon a set of objects that is different from the natural ordering provided for by the implementation of the `Comparable` interface.

The `Comparator` interface declares two methods: 1) `compare(Object o1, Object o2)`, and 2) `equals(Object o)`. The `compare` method compares the two argument objects `o1` and `o2`. Just like the `Comparable`'s `compareTo()` method, the `compare()` method should perform consistently with the `equals()` method. However, in this example, I will create a comparator class that can be used to sort `Person` objects by age. Example 23.12 gives the code for the `PersonAgeComparator` class.

23.12 *PersonAgeComparator.java*

```

1      import java.util.*;
2      import java.io.*;
3
4      public final class PersonAgeComparator implements Comparator, Serializable {
5          public final int compare(Object o1, Object o2){
6              int result = 0;
7              if(((Person)o1).getAge() < ((Person)o2).getAge())

```



```

Terminal - tcsh
[rick-millers-computer:~/desktop/overriding_object_methods] swodog$ java CollectionTestApp
Rick W Miller Male 6/5/1963
Debbie S Sears Female 3/8/1986
John L Thompson Male 4/1/1978
Riddic H Bean Male 15/11/2001
Gloria J Albright Female 2/8/1923
Zena P Warrior Female 26/1/1968
Shelly H Marshall Female 15/7/1993
Jessica R Simpson Female 3/6/1912
Peter R Rabbit Male 30/4/1999
Mohamad R Abbas Male 29/5/1961
Sapna P Gupta Female 11/8/1988
Marvin D Williams Male 18/2/1945
Kyle O Miller Male 13/9/1954
Joseph L Smith Male 23/10/1963
Nana G Jones Female 4/11/1977
Hedy E Lamarr Female 26/2/1914
-----Sorted List-----
Debbie S Sears Female 3/8/1986
Gloria J Albright Female 2/8/1923
Hedy E Lamarr Female 26/2/1914
Jessica R Simpson Female 3/6/1912
John L Thompson Male 4/1/1978
Joseph L Smith Male 23/10/1963
Kyle O Miller Male 13/9/1954
Marvin D Williams Male 18/2/1945
Mohamad R Abbas Male 29/5/1961
Nana G Jones Female 4/11/1977
Peter R Rabbit Male 30/4/1999
Rick W Miller Male 6/5/1963
Riddic H Bean Male 15/11/2001
Sapna P Gupta Female 11/8/1988
Shelly H Marshall Female 15/7/1993
Zena P Warrior Female 26/1/1968
[rick-millers-computer:~/desktop/overriding_object_methods] swodog$

```

Figure 23-7: Results of Running Example 23.11

```

7         result = -1;
8         else if(((Person)o1).getAge() == ((Person)o2).getAge())
9             result = 0;
10        else if(((Person)o1).getAge() > ((Person)o2).getAge())
11            result = 1;
12        return result;
13    }
14
15    public final boolean equals(Object o){
16        if(o == null) return false;
17        boolean is_equal = false;
18        if(o instanceof PersonAgeComparator){
19            is_equal = true;
20        }
21        return is_equal;
22    }
23 } // end PersonAgeComparator

```

Referring to example 23.12 — the class is declared to be final so that it cannot be extended. It also implements the `Serializable` interface as recommended in the Java Platform API documentation. This is recommended because the comparator may be used to order serializable data structures. (*If the collection is serialized, the comparator used to order the collection must be serialized along with it.*) The `compare()` method takes two `Object` references as arguments. It casts each to the `Person` type and compares their ages using their `getAge()` methods. Now, let's see if this thing works! Example 23.13 gives a modified version of the `CollectionTestApp` program.

23.13 *CollectionTestApp.java (mod 1)*

```

1     import java.util.*;
2
3     public class CollectionTestApp {
4         public static void main(String[] args){
5             List list = new ArrayList();
6
7             list.add(new Person("Rick", "W", "Miller", 1963, 5, 6, Person.MALE));
8             list.add(new Person("Debbie", "S", "Sears", 1986, 8, 3, Person.FEMALE));
9             list.add(new Person("John", "L", "Thompson", 1978, 1, 4, Person.MALE));
10            list.add(new Person("Riddic", "H", "Bean", 2001, 11, 15, Person.MALE));
11            list.add(new Person("Gloria", "J", "Albright", 1922, 12, 2, Person.FEMALE));
12            list.add(new Person("Zena", "P", "Warrior", 1968, 1, 26, Person.FEMALE));
13            list.add(new Person("Shelly", "H", "Marshall", 1993, 7, 15, Person.FEMALE));
14            list.add(new Person("Jessica", "R", "Simpson", 1912, 6, 3, Person.FEMALE));
15            list.add(new Person("Peter", "R", "Rabbit", 1999, 4, 30, Person.MALE));

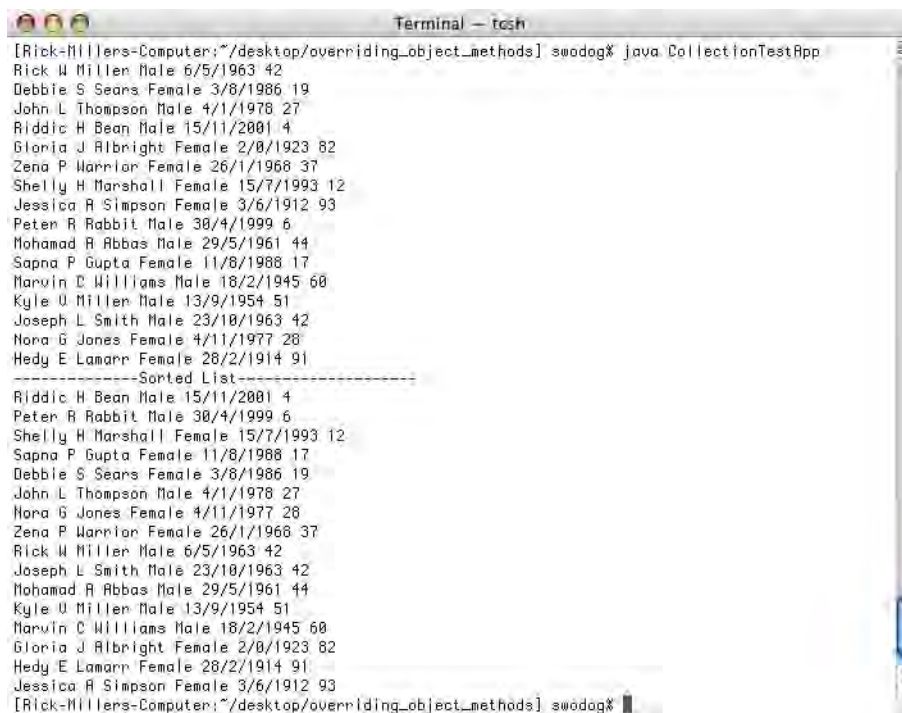
```

```

16     list.add(new Person("Mohamad", "A", "Abbas", 1961, 5, 29, Person.MALE));
17     list.add(new Person("Sapna", "P", "Gupta", 1988, 8, 11, Person.FEMALE));
18     list.add(new Person("Marvin", "C", "Williams", 1945, 2, 18, Person.MALE));
19     list.add(new Person("Kyle", "V", "Miller", 1954, 9, 13, Person.MALE));
20     list.add(new Person("Joseph", "L", "Smith", 1963, 10, 23, Person.MALE));
21     list.add(new Person("Nora", "G", "Jones", 1977, 11, 4, Person.FEMALE));
22     list.add(new Person("Hedy", "E", "Lamarr", 1914, 2, 28, Person.FEMALE));
23
24     for(Iterator i = list.iterator(); i.hasNext();){
25         Person p = (Person)i.next();
26         System.out.println(p.toString() + " " + p.getAge());
27     }
28
29     Collections.sort(list, new PersonAgeComparator());
30
31     System.out.println("-----Sorted List-----");
32     for(Iterator i = list.iterator(); i.hasNext();){
33         Person p = (Person)i.next();
34         System.out.println(p.toString() + " " + p.getAge());
35     }
36 }
37 } // end CollectionTestApp

```

Referring to example 23.13 — a few modifications were made to the program. The format of the console output was changed slightly to include age information in addition to the standard `Person.toString()` text. The primary difference, however, is the use of the `PersonAgeComparator` class on line 29. A new `PersonAgeComparator` object is created using the new operator and used as the second argument to the `Collections.sort()` method. This causes the list of `Person` objects to be sorted according to age. Figure 23-8 gives the results of running this program.



```

Terminal - tcsh
[rick@millers-Computer:~/desktop/overriding_object_methods] swadog$ java CollectionTestApp
Rick W Miller Male 6/5/1963 42
Debbie S Sears Female 3/8/1986 19
John L Thompson Male 4/1/1978 27
Riddic H Bean Male 15/11/2001 4
Gloria J Albright Female 2/8/1923 82
Zena P Warrior Female 26/1/1968 37
Shelly H Marshall Female 15/7/1993 12
Jessica R Simpson Female 3/6/1912 93
Peter R Rabbit Male 30/4/1999 6
Mohamad A Abbas Male 29/5/1961 44
Sapna P Gupta Female 11/8/1988 17
Marvin C Williams Male 18/2/1945 60
Kyle W Miller Male 13/9/1954 51
Joseph L Smith Male 23/10/1963 42
Nora G Jones Female 4/11/1977 28
Hedy E Lamarr Female 28/2/1914 91
-----Sorted List-----
Riddic H Bean Male 15/11/2001 4
Peter R Rabbit Male 30/4/1999 6
Shelly H Marshall Female 15/7/1993 12
Sapna P Gupta Female 11/8/1988 17
Debbie S Sears Female 3/8/1986 19
John L Thompson Male 4/1/1978 27
Nora G Jones Female 4/11/1977 28
Zena P Warrior Female 26/1/1968 37
Rick W Miller Male 6/5/1963 42
Joseph L Smith Male 23/10/1963 42
Mohamad A Abbas Male 29/5/1961 44
Kyle W Miller Male 13/9/1954 51
Marvin C Williams Male 18/2/1945 60
Gloria J Albright Female 2/8/1923 82
Hedy E Lamarr Female 28/2/1914 91
Jessica R Simpson Female 3/6/1912 93
[rick@millers-Computer:~/desktop/overriding_object_methods] swadog$

```

Figure 23-8: Results of Running Example 23.13

Quick Review

The `Comparator` interface is implemented by a class of objects whose job it is to impose a total ordering upon another class of objects. It is suggested that the `Comparator.compare()` method's behavior be consistent with the `equals()` method of the class upon which it is imposing order.

IMPLEMENTING THE SERIALIZABLE INTERFACE

Since I discussed the Serializable interface in detail in chapter 18 and offered a practical example again in chapter 21 I will keep this section brief. I just want to remind you that if you intend to serialize a class of objects the class must implement the java.io.Serializable interface. (*I think this is the shortest section in this book!*)

PERSON CLASS – THE FINAL VERSION

Example 23.14 gives the code for the final version of the Person class. In its current state it can be expected to perform predictably in many application contexts.

23.14 Person.java (Final Version)

```

1      import java.util.*;
2      import java.io.*;
3
4      public class Person implements Cloneable, Comparable, Serializable {
5          private String first_name = null;
6          private String middle_name = null;
7          private String last_name = null;
8          private Calendar birthday = null;
9          private String gender = null;
10
11         public static final String MALE = "Male";
12         public static final String FEMALE = "Female";
13
14         public Person(String f_name, String m_name, String l_name, int dob_year, int dob_month,
15             int dob_day, String gender){
16             first_name = f_name;
17             middle_name = m_name;
18             last_name = l_name;
19             this.gender = gender;
20
21             birthday = Calendar.getInstance();
22             birthday.set(dob_year, dob_month, dob_day);
23         }
24
25         public int getAge(){
26             Calendar today = Calendar.getInstance();
27             int now = today.get(Calendar.YEAR);
28             int then = birthday.get(Calendar.YEAR);
29             return (now - then);
30         }
31
32         public String getFullName(){ return (first_name + " " + middle_name + " " + last_name); }
33
34         public String getFirstName(){ return first_name; }
35         public void setFirstName(String f_name) { first_name = f_name; }
36
37         public String getMiddleName(){ return middle_name; }
38         public void setMiddleName(String m_name){ middle_name = m_name; }
39
40         public String getLastName(){ return last_name; }
41         public void setLastName(String l_name){ last_name = l_name; }
42
43         public String getNameAndAge(){ return (getFullName() + " " + getAge()); }
44
45         public String getGender(){ return gender; }
46         public void setGender(String gender){ this.gender = gender; }
47
48         public void setBirthday(int year, int month, int day){ birthday.set(year, month, day); }
49
50         public String toString(){
51             return this.getFullName() + " " + gender + " " + birthday.get(Calendar.DATE) + "/"
52                 + birthday.get(Calendar.MONTH) + "/" + birthday.get(Calendar.YEAR);
53         }
54
55         public boolean equals(Object o){
56             if(o == null) return false;
57             boolean is_equal = false;
58             if(o instanceof Person){
59                 if(this.first_name.equals(((Person)o).first_name) &&

```

```

60         this.middle_name.equals(((Person)o).middle_name) &&
61         this.last_name.equals(((Person)o).last_name) && this.gender.equals(((Person)o).gender) &&
62         (this.birthday.get(Calendar.YEAR) == ((Person)o).birthday.get(Calendar.YEAR)) &&
63         (this.birthday.get(Calendar.MONTH) == ((Person)o).birthday.get(Calendar.MONTH)) &&
64         (this.birthday.get(Calendar.DATE) == ((Person)o).birthday.get(Calendar.DATE)) ) {
65             is_equal = true;
66         }
67     }
68     return is_equal;
69 }
70
71 public int hashCode(){
72     return this.toString().hashCode();
73 }
74
75 public Object clone() throws CloneNotSupportedException {
76     super.clone();
77     return new Person(new String(first_name), new String(middle_name), new String(last_name),
78         birthday.get(Calendar.YEAR), birthday.get(Calendar.MONTH), birthday.get(Calendar.DATE),
79         new String(gender));
80 }
81
82 public int compareTo(Object o){
83     return this.toString().compareTo(o.toString());
84 }
85
86 } //end Person class

```

SUMMARY

Anticipated object usage scenarios must be considered during the application design phase. All classes should be evaluated against the seven object usage scenarios. Use the Object Usage Scenario Evaluation Checklist given in table 23-1 to help evaluate your class design.

Most of the methods provided by the `java.lang.Object` class are meant to be overridden. These include the `toString()`, `equals()`, `hashCode()`, `clone()`, and `finalize()` methods.

The purpose of the `toString()` method is to provide a string representation for a class of objects. The contents of this string are dictated by class design requirements. It's considered good programming practice to always override the `toString()` method.

The purpose of the `equals()` method is to perform a logical equality comparison between two objects of the same class. The `equals()` method implements an equivalence relation specified in the Java Platform API documentation that says the method should be reflexive, symmetric, transitive, consistent, and should return false if a non-null instance object is compared against a null reference. The behavior of the `equals()` method has ramifications for the behavior of other methods as well.

The purpose of the `hashCode()` method is to provide an integer value for an object when it is used with hashtable or hash-based data structures. The `hashCode()` method has a general behavior contract specified in the Java Platform API documentation and its performance is dependent upon the behavior of the `equals()` method. Namely, fields utilized in the `equals()` method to perform the logical comparison should be used in the `hashCode()` method to generate the object's hash code.

The purpose of the `clone()` method is to create a logically equal copy of an existing object. The important thing to consider when implementing the `clone()` method is the undesirable effects of performing a shallow copy of complex class types.

The purpose of the `finalize()` method is to tie-up loose ends when an object is collected by the Java Virtual Machine (JVM) garbage collector. However, do not depend on `finalize()` to perform a time-sensitive operation as there are no guarantees when the `finalize()` method will be called.

Implement the `Comparable` interface when a class of objects will be utilized in a collection and will therefore be subject to a natural ordering. It is recommended that the `Comparable.compareTo()` method perform consistently with the `equals()` method.

The `Comparator` interface is implemented by a class of objects whose job it is to impose a total ordering upon another class of objects. It is suggested that the `Comparator.compare()` method's behavior be consistent with the `equals()` method of the class upon which it is imposing order.

Implement the `Serializable` interface if you need to save objects to disk or send them over a network.

Skill-Building Exercises

1. **Reasoning About Design:** Given the special treatment of String objects in the Java environment, why do you suppose Ashmore’s simple hash code algorithm works?
2. **Creating Different Comparators:** Create several different Comparator classes that can be used to sort Person objects by last name and birthday.
3. **Testing The hashCode() Method:** Write a short program that generates ten thousand logically equal Person objects and see if their hash codes are the same.
4. **Bloch’s Hash Code Algorithm:** Implement Bloch’s hash code algorithm in the body the Person hashCode() method and repeat skill-building exercise 3 above.
5. **Reasoning About Design:** What effects, if any, would different implementations of the Person.toString() method have on the performance of the Person.compareTo() method?
6. **Apples And Oranges:** Create an array of Strings with element values of “1”, “2”, “3”, “9”, “10”, “18”, “22”, “34”. Next, create an array of integers (*primitive type ints*) and populate the array with the following element values: 1, 2, 3, 9, 10, 18, 22, 34. Sort both arrays using the Arrays.sort() method and explain the results.
7. **Generic Methods:** Convert the Person.equals(), Person.clone(), and Person.compareTo() methods to Java 5 generic methods.
8. **Generic Class:** Convert the PersonAgeComparator class to a Java 5 generic class that operates on Person class objects or subtypes.

SUGGESTED PROJECTS

1. **Employee Class Revisited:** Revise the Employee class given in chapter 22, example 22.2 to utilize the new and improved Person class presented in this chapter. Employ the Object Usage Scenario Checklist and take the necessary steps to ensure that HourlyEmployee and SalariedEmployee derived class objects behave well in all seven usage scenarios.
2. **Create Comparator Classes For Employee Objects:** Create several Comparator classes that impose ordering on Employee objects. Create comparators that can be used to sort employees by employee number, age, and last name. Reuse as much functionality provided by the Person class as possible.

SELF-TEST QUESTIONS

1. Why do you think it is important for objects to behave well?
2. What are the seven object usage scenarios you should be aware of when designing a class?
3. What five methods of the java.lang.Object class are most often overridden?
4. What is the purpose of the toString() method?

5. What is the purpose of the equals() method?
6. What are the five elements of the equals() method equivalence relation as stated in the Java Platform API documentation?
7. What is the purpose of the hashCode() method?
8. What is the general contract of the hashCode() method as stated in the Java Platform API documentation?
9. How does a class's hashCode() method behavior depend upon the behavior provided by the class's equals() method?
10. What type of ordering does the Comparable.compareTo() method impose upon a class of objects?

REFERENCES

Joshua Bloch. *Effective Java™ Programming Language Guide*. Addison-Wesley, Boston, MA. ISBN: 0-201-31005-8.

Java 5 Platform API Documentation: [<http://java.sun.com/j2se/1.5.0/docs/api/index.html>]

David Flanagan. *Java™ In A Nutshell*, Fifth Edition. O'Reilly Media, Inc. Sebastopol, CA. ISBN: 0-596-00773-6.

NOTES

CHAPTER 24



Flowers

THREE DESIGN PRINCIPLES

LEARNING OBJECTIVES

- *LIST THE PREFERRED CHARACTERISTICS OF AN OBJECT-ORIENTED APPLICATION ARCHITECTURE*
- *STATE THE DEFINITION OF THE LISKOV SUBSTITUTION PRINCIPLE (LSP)*
- *STATE THE DEFINITION OF BERTRAND MEYER'S DESIGN BY CONTRACT PROGRAMMING (DbC)*
- *DESCRIBE THE CLOSE RELATIONSHIP BETWEEN THE LISKOV SUBSTITUTION PRINCIPLE AND DESIGN BY CONTRACT*
- *STATE THE PURPOSE OF METHOD PRECONDITIONS AND POSTCONDITIONS*
- *STATE THE PURPOSE OF CLASS INVARIANTS*
- *DESCRIBE THE EFFECTS WEAKENING AND STRENGTHENING PRECONDITIONS HAVE ON SUBCLASS BEHAVIOR*
- *DESCRIBE THE EFFECTS WEAKENING AND STRENGTHENING POSTCONDITIONS HAVE ON SUBCLASS BEHAVIOR*
- *STATE THE PURPOSE AND USE OF THE OPEN-CLOSED PRINCIPLE*
- *STATE THE PURPOSE AND USE OF THE DEPENDENCY INVERSION PRINCIPLE*
- *DESCRIBE THE CHARACTERISTICS OF BAD SOFTWARE ARCHITECTURE*
- *DEMONSTRATE YOUR ABILITY TO USE THE THREE DESIGN PRINCIPLES IN YOUR PROGRAMMING PROJECTS*

INTRODUCTION

Building complex, well-behaved, object-oriented software is a difficult task for several reasons. First, simply programming in Java does not automatically make your application object-oriented. Second, the process by which you become proficient at object-oriented design and programming is characterized by experience. It takes a lot of time to learn the lessons of bad software architecture design and apply those lessons learned to create good object-oriented architectures.

The objective of this chapter is to help you jump-start your object-oriented architectural design efforts. I begin with a discussion of the preferred characteristics of a well-designed object-oriented architecture. I then present and discuss three important object-oriented design principles that you can immediately apply to your software architecture designs to drastically improve performance, reliability, and maintainability.

The three design principles include the Liskov Substitution Principle (LSP), the Open-Closed Principle (OCP), and the Dependency Inversion Principle (DIP). Bertrand Meyer's Design by Contract (DbC) programming is discussed in the context of its close relationship to, and extension of, the Liskov Substitution Principle.

An understanding of these three design principles, coupled with an understanding of how to apply them using the Java programming language, will significantly improve your ability to design robust, object-oriented software architectures.

THE PREFERRED CHARACTERISTICS OF AN OBJECT-ORIENTED ARCHITECTURE

From a programmer's perspective, a well-designed, object-oriented architecture manifests itself as an inheritance hierarchy, including a set of abstract data type vertical and horizontal (*compositional*) relationships, that exhibits several key characteristics. It is 1) easy to understand, 2) easy to reason about, and 3) easy to extend. These characteristics are discussed briefly below.

EASY TO UNDERSTAND – (HOW DOES THIS THING WORK?)

A programmer, when shown a component diagram of a complex software system, should be able to understand what it does, or what it is you are trying to do, in about five minutes flat. To do this a software architecture must be designed to be understood.

The organizational complexity of large software systems can be overwhelming if the architecture is poorly designed. An application comprised of even a small number of tightly coupled software components requires significantly more effort to understand than one designed to be understood quickly. An application software architecture must be thoroughly understood by a programmer before the effects of changing its components or adding functionality can be accurately assessed.

EASY TO REASON ABOUT – (WHAT ARE THE EFFECTS OF CHANGE?)

The effects of changing pieces of a software application must be fully predictable. Programmers must be confident that the changes they make to one code module will not mysteriously break another, seemingly unrelated, module in the system. If the effects of change can be accurately predicted then the architecture can be reasoned about. The best way to reason about the effects of change is to render code changes unnecessary. (*The effects of no change is definitely predictable!*)

EASY TO EXTEND – (WHERE DO I ADD FUNCTIONALITY?)

Well-designed application architectures accommodate the addition of features and facilitate component reuse. A programmer, when tasked with adding new functionality to an application, must know exactly where to put it. The act of adding functionality should not require the changing of existing code, but rather its extension.

THE LISKOV SUBSTITUTION PRINCIPLE & DESIGN BY CONTRACT

Dr. Barbara Liskov and Dr. Bertrand Meyer are both important figures in the object-oriented software research community. The two design principles and guidelines that bear their name are the Liskov Substitution Principle (LSP) and Bertrand Meyer's Design by Contract (DbC). These closely related object-oriented design concepts are covered together in this section and can be summarized in the following statement:

Subtype objects must be behaviorally substitutable for supertype objects. Programmers must be able to reason correctly about and rely upon the behavior of subtypes using only the supertype behavior specification.

REASONING ABOUT THE BEHAVIOR OF SUPERTYPES AND SUBTYPES

Programmers must be able to reason correctly about the behavior of abstract data types and their derived subtypes. The LSP and DbC provide both theoretical and applied foundations upon which programmers can build well-behaved class inheritance hierarchies that facilitate the object-oriented architectural reasoning process.

Relationship BETWEEN THE LSP AND DbC

The LSP and DbC are closely related concepts primarily because they both draw from largely the same body of research in the formulation of their theories. They each address the question of how a programmer should be able to reason about the behavior of a subtype object when it is substituted for a supertype object, they each address the role of method *preconditions* and *postconditions* in the specification of desired object behavior, and they each discuss the role of *class invariants* and how method postconditions should ensure invariant state conditions are preserved. *They both seek to provide a mechanism for programmers to create reliable object-oriented software.*

Design by contract differs from the LSP in its emphasis on the notion of contracts between supertype and subtype. The base class (*supertype*) is a contractor that may, at runtime, have its interface methods performed by a sub-contractor (*subtype*). Programmers should not need any apriori knowledge of the subtype's existence when they write the code that may come to rely on the subtype's behavior. The subtype, when substituted for the supertype, should fulfill the contract promised by the supertype. In other words, the subtype object should not pull any surprises.

Another difference between the LSP and DbC is that the LSP is more notional, while DbC is more practical. By this I mean no language, as of this writing, directly supports the LSP specifically, with perhaps the exception of the type checking facilities provided by a compiler. Design by contract, on the other hand, is directly supported by the Eiffel programming language.

THE COMMON GOAL OF THE LSP AND DbC

The LSP and DbC share a common goal. They both aim to help software developers build correct software from the start. Given this common goal I will occasionally refer to both concepts collectively as the LSP/DbC.

JAVA SUPPORT FOR THE LSP AND DbC

With the exception of type checking, Java does not provide direct language support for either the LSP or DbC. However, there are techniques you can use to enforce preconditions and postconditions, and to ensure the state of class invariants. Regardless of the level of language support for either the LSP or DbC, programmers can realize significant improvements in their overall class hierarchy designs by simply keeping the LSP and DbC in mind during the design process.

DESIGNING WITH THE LSP/DbC IN MIND

The LSP/DbC focuses on the correct specification of supertype and subtype behavioral relationships. By keeping the LSP/DbC in mind when designing class hierarchies programmers are much less likely to create subclasses that implement behavior incompatible with that specified by the base class.

CLASS DECLARATIONS VIEWED AS BEHAVIOR SPECIFICATIONS

A class declaration introduces a new abstract data type into a programmer's environment. The class declaration is, by its very nature, a behavioral specification. The behavior is specified by the set of public interface methods made available to clients, by the set of possible states an object may assume, and by the side effects resulting from method execution.

In Java, class declaration and definition is usually combined. A class that specifies behavior only is known in Java as an interface whereas an abstract class can both specify behavior, and, where necessary, provide behavior implementation.

An abstract data type can adopt the behavioral specification of another abstract data type. (*Like one interface extending another interface, or a class extending another class.*) The former would be the subtype and the latter the supertype. When the supertype is an abstract base class or interface, the subtype inherits only a behavior specification. It must then either implement the specified behavior or further defer the implementation to yet another subtype. When a supertype provides behavior implementation, a subtype may adopt the supertype behavior outright or provide an overriding behavior. *It is the correct implementation of this overriding behavior about which the LSP/DbC is most concerned.* Programmers can create well-behaved subtypes by employing preconditions, postconditions, and class invariants.

Quick REVIEW

The Liskov Substitution Principle (LSP) and Bertrand Meyer's Design by Contract (DbC) programming are closely related principles designed to enable programmers to better reason about subtype behavior.

PRECONDITIONS, POSTCONDITIONS, AND CLASS INVARIANTS

Preconditions, postconditions, and class invariants are the three cornerstones of both the LSP and DbC. Their definitions and application are discussed in this section.

CLASS INVARIANT

A class invariant is an assertion about an object property that must hold true for all valid states an object can assume. For example, suppose an airplane object has a speed property that can be set to a range of integer values between 0 and 800. This rule should be enforced for all valid states an airplane object can assume. All methods that can be invoked on an airplane object must ensure they do not set the speed property to less than 0 or greater than 800.

PRECONDITION

A precondition is an assertion about some condition that must be true before a method can be expected to perform its operation correctly. For example, suppose the airplane object's speed property can be incremented by some value and there exists in the set of airplane's public interface methods one that increments the speed property anywhere from 1 to 5 depending on the value of the argument supplied to the method. For this method to perform correctly, it must check that the argument is in fact a valid increment value of 1, 2, 3, 4, or 5. If the increment value tests valid then the precondition holds true and the increment method should perform correctly.

The precondition must be true before the method is called, therefore it is the responsibility of the caller to make the precondition true, and the responsibility of the called method to enforce the truth of the precondition.

POSTCONDITION

A postcondition is an assertion that must hold true when a method completes its operations and returns to the caller. For example, the airplane's speed increment method should ensure that the class invariant speed property being $0 \leq \text{speed} \leq 800$ holds true when the increment method completes its operations.

AN EXAMPLE

Example 24.1 gives the code for a class named `Incrementer`. An `incrementer` object can simply be incremented by the values 1, 2, 3, 4, or 5 and maintain a state value between 0 and 100. This example illustrates one approach to enforcing method preconditions and postconditions via the Java assertion mechanism.

24.1 *Incrementer.java*

```

1      public class Incrementer {
2          /*****
3              Class invariant: 0 <= Incrementer.val <= 100
4              *****/
5          private int val = 0;
6
7          /*****
8              Constructor Method: Incrementer(int i)
9              precondition: ((0 <= i) && (i <= 100))
10             postcondition: 0 <= Incrementer.val <= 100
11             *****/
12         public Incrementer(int i){
13             assert((0 <= i) && (i <= 100));
14             val = i;
15             System.out.println("Incrementer object created with initial value of: " + val);
16             checkInvariant(); // enforce class invariant
17         }
18
19         /*****
20             Method: void increment(int i)
21             precondition: 0 < i <= 5
22             postcondition: 0 <= Incrementer.val <= 100
23             *****/
24         public void increment(int i){
25             assert((0 < i) && (i <= 5)); // enforce precondition
26
27             if((val+i) <= 100){
28                 val += i;
29             }else{
30                 int temp = val;
31                 temp += i;
32                 val = (temp - 100);
33             }
34
35             checkInvariant(); // enforce class invariant
36
37             System.out.println("Incrementer value is: " + val);
38         }
39
40         /*****
41             Method: void checkInvariant() - called
42             immediately after any change to class
43             invariant to ensure invariant condition
44             is satisfied.
45             *****/
46         private void checkInvariant(){
47             assert((0 <= val) && (val <= 100));
48         }
49     } // end Incrementer class definition

```

Referring to example 24.1 — the `Incrementer` class has a private instance field of type `int` named `val`. The class invariant is specified in comments above the field declaration and indicates that the valid range of values `val` can assume is 0 through 100. This invariant is enforced with the Java assertion mechanism in the body of the constructor when an instance of `Incrementer` is created. The invariant is also validated via the `checkInvariant()` method.

The `increment()` method's pre- and postconditions are stated in the comment above the method. It indicates that the valid range of increment values the parameter `i` can assume include 1 through 5, and that when the method com-

pletes the class invariant state must be valid. The `increment()` method's precondition is checked by the `assert` statement on line 25. The class invariant is checked by calling the `checkInvariant()` method on line 35.

To compile this class using Java version 1.4 you must inform the Java compiler to generate Java 1.4 compatible source code using the `-source` switch in the following manner:

```
javac -source 1.4 Incrementer.java
```

Example 24.2 gives a short program putting the `Incrementer` class through its paces.

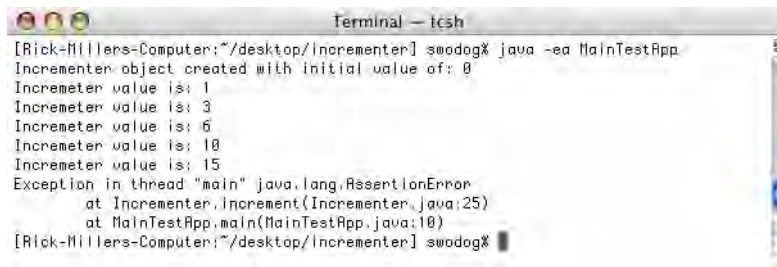
24.2 *MainTestApp.java*

```
1     public class MainTestApp {
2         public static void main(String[] args){
3             Incrementer i1 = new Incrementer(0);
4
5             i1.increment(1);
6             i1.increment(2);
7             i1.increment(3);
8             i1.increment(4);
9             i1.increment(5);
10            i1.increment(6); // throws an AssertionError exception as expected
11
12        } // end main() method
13    } // end MainTestApp clas definition
```

Referring to example 24.2 — an `Incrementer` reference named `i1` is declared and initialized on line 3. On lines 5 through 10 the `increment` method is called via `i1` with different increment values 1, 2, 3, 4, 5, and 6. If assertions have been enabled in the JVM when this example is executed then it will throw an `AssertionError` exception when line 10 executes. To enable assertions in the JVM start the `MainTestApp` program using the `enable assertions (-ea)` switch in the following manner:

```
java -ea MainTestApp
```

Figure 24-1 shows the results of running this program.



```
Terminal — tcsh
[rick-millers-computer:~/desktop/incrementer] swadog% java -ea MainTestApp
Incrementer object created with initial value of: 0
Incrementer value is: 1
Incrementer value is: 3
Incrementer value is: 6
Incrementer value is: 10
Incrementer value is: 15
Exception in thread "main" java.lang.AssertionError
    at Incrementer.increment(Incrementer.java:25)
    at MainTestApp.main(MainTestApp.java:10)
[rick-millers-computer:~/desktop/incrementer] swadog%
```

Figure 24-1: Results of Running Example 24.2

A NOTE ON USING ASSERTIONS TO ENFORCE PRE- AND POSTCONDITIONS

As was just demonstrated, the `assert` keyword is ignored in Java version 1.4 unless the `-source` switch is used during compilation. Also, assertions are not enabled in the JVM unless it is started with the `-ea` switch. The use of the assertion mechanism to enforce method pre- and postconditions, and the state of class invariants, is best used during implementation and testing. Remember, it is the responsibility of the calling program to adhere to a method's documented precondition.

Consider for a moment the `MainTestApp` program shown in example 24.1. When a programmer runs this code and gets the error produced by trying to call the `increment` method with an invalid precondition, he would then be obliged to fix his code to eliminate the error. From this point forward the assertion mechanism can be safely disabled and the code will run fine.

USING INCREMENTER AS A BASE CLASS

A programmer using the `Incrementer` class learns from reading its class invariant, precondition, and postcondition specifications how those objects can be used in a program and how they should behave. And that is all they

should have to know, even when an `Incrementer` reference points to an object belonging to a class that is derived from `Incrementer`.

There are several issues that demand the attention of the programmer who plans to extend the functionality of `Incrementer`. First, he must be aware of the point of view of the client program that will use the derived object. That code expects certain behavior from `Incrementer` objects. For example, a client program calling the `increment()` method on `Incrementer` objects can rely on proper behavior *if* the arguments to the method satisfy the precondition of being greater than zero or less than or equal to five. If an object derived from `Incrementer` is substituted at runtime for an `Incrementer` object, the derived object must not break the client code by behaving in a manner not anticipated by the client program.

Second, with the expectations of the client code in mind, what rules should a programmer follow when extending the functionality of a base class to ensure the derived object continues to live up to or meet the expectations of the client code? This section explores these issues further.

Example 24.3 gives the code for a class named `DerivedIncrementer` that extends the functionality of the `Incrementer` class.

24.3 *DerivedIncrementer.java*

```

1      public class DerivedIncrementer extends Incrementer {
2          /*****
3              Class invariant: 0 <= val <= 50
4          *****/
5          private int val = 0;
6
7          /*****
8              Constructor Method: DerivedIncrementer(int i)
9              precondition: ((0 <= i) && (i <= 50))
10             postcondition: 0 <= val <= 50
11          *****/
12         public DerivedIncrementer(int i){
13             super(i);
14             assert((0 <= i) && (i <= 50)); // enforce precondition
15             val = i;
16             System.out.println("DerivedIncrementer object created with value: " + val);
17             checkInvariant();
18         }
19
20         /*****
21             Method: void increment(int i)
22             precondition: ((0 < i) && (i <= 5))
23             postcondition: 0 <= val <= 50
24          *****/
25         public void increment(int i){
26             assert((0 < i) && (i <= 5)); // enforce precondition
27             super.increment(i);
28             if((val+i) <= 50){
29                 val += i;
30             }else{
31                 int temp = val;
32                 temp += i;
33                 val = (temp - 50);
34             }
35             checkInvariant(); // check invariant
36             System.out.println("DerivedIncrementer value is: " + val);
37         }
38
39         private void checkInvariant(){
40             assert((0 <= val) && (val <= 50));
41         }
42     } // end DerivedIncrementer class definition

```

Referring to example 24.3 — the `DerivedIncrementer` class extends `Incrementer` and overrides its `increment()` method. `DerivedIncrementer` has its own `val` field which has a different class invariant from that of `Incrementer`'s `val`. (*But this is perfectly OK!*) The `DerivedIncrementer`'s version of the `increment()` method subscribes to the same precondition as that of the base class version of the method it is overriding, namely, that the values of the integer parameter `i` can be anything from 0 through 5. Therefore, an object of type `DerivedIncrementer` will behave the same as objects of type `Incrementer`. Example 24.4 shows the `DerivedIncrementer` class in action.

24.4 *MainTestApp.java (mod 1)*

```

1      public class MainTestApp {
2          public static void main(String[] args){
3              Incrementer i1 = new Incrementer(0);
4              Incrementer i2 = new DerivedIncrementer(20);
5          }

```

```

6         i1.increment(1);
7         i1.increment(2);
8         i1.increment(3);
9         i1.increment(4);
10        i1.increment(5);
11        System.out.println("-----");
12        i2.increment(4);
13        i2.increment(5);
14        i2.increment(6); // throws AssertionError exception here as expected
15
16    } // end main() method
17 } // end MainTestApp clas definition

```

```

Terminal — tcsh
[rick@millers-Computer:~/desktop/incrementer] swodog$ java -ea MainTestApp
Incrementer object created with initial value of: 0
Incrementer object created with initial value of: 20
DerivedIncrementer object created with value: 20
Incrementer value is: 1
Incrementer value is: 3
Incrementer value is: 6
Incrementer value is: 10
Incrementer value is: 15
-----
Incrementer value is: 24
DerivedIncrementer value is: 24
Incrementer value is: 29
DerivedIncrementer value is: 29
Exception in thread "main" java.lang.AssertionError
    at DerivedIncrementer.increment(DerivedIncrementer.java:26)
    at MainTestApp.main(MainTestApp.java:14)
[rick@millers-Computer:~/desktop/incrementer] swodog$

```

Figure 24-2: Results of Running Example 24.4

Referring to example 24.4 — an `Incrementer` type reference named `i2` is declared on line 4 and initialized to point to an object of type `DerivedIncrementer`. From lines 12 through 14 the `increment()` method is called on the `DerivedIncrementer` object via `i2`. When the invalid precondition value of 6 is used in the `increment()` method call the `AssertionError` is thrown as expected. Figure 24-2 shows the results of running this program.

CHANGING THE PRECONDITIONS OF DERIVED CLASS METHODS

The version of the `increment()` method in class `DerivedIncrementer` discussed above implemented the same precondition as the `Incrementer` class version, namely, that the integer argument passed to the method was in the range 1 through 5. However, it is possible to specify a different precondition for a derived class version of `increment()`.

In regards to derived class method preconditions you can go three ways: 1) adopt the same precondition(s), as was illustrated in the previous section, 2) weaken the precondition(s), or 3) strengthen the precondition(s).

ADOPTING THE SAME PRECONDITIONS

Derived class methods can adopt the same preconditions as the base class methods they override. The `increment()` method in class `DerivedIncrementer`, shown in the previous section, adopted the same precondition as the `Incrementer` class's version of `increment()`. When a derived class method adopts the same preconditions as its base class counterpart its behavior is predictable from the point of view of any client program using a base class reference to a derived class object. In other words, you can safely reason about the behavior of a derived class object whose overriding methods adopt the same preconditions as their base class counterparts.

WEAKENING PRECONDITIONS

Derived class methods can weaken the preconditions specified in the base class methods they override. Weakening can also be thought of as a loosening or relaxing of a specified precondition. The `increment()` method in class `DerivedIncrementer` could have weakened the precondition specified in the base class version of `increment()` by allowing a wider range of increment values to be called as arguments. An example of this is shown in the class named `WeakenedDerivedIncrementer` whose code is given in example 24.5.

24.5 *WeakenedDerivedIncrementer.java*

```

1      public class WeakenedDerivedIncrementer extends Incrementer {
2          /*****
3              Class invariant: 0 <= val <= 50
4              *****/
5          private int val = 0;
6
7          /*****
8              Constructor Method: DerivedIncrementer(int i)
9              precondition: ((0 <= i) && (i <= 50))
10             postcondition: 0 <= val <= 50
11             *****/
12         public WeakenedDerivedIncrementer(int i){
13             super(i);
14             assert((0 <= i) && (i <= 50)); // enforce precondition
15             val = i;
16             System.out.println("WeakenedDerivedIncrementer object created with value: " + val);
17             checkInvariant();
18         }
19
20         /*****
21             Method: void increment(int i)
22             precondition: ((0 < i) && (i <= 10))
23             postcondition: 0 <= val <= 50
24             *****/
25         public void increment(int i){
26             assert((0 < i) && (i <= 10)); // enforce precondition
27
28             if((0 <= i) && (i <= 5)){ // remember, it's our job to use the base class correctly!
29                 super.increment(i);
30             }
31
32             if((val+i) <= 50){
33                 val += i;
34             }else{
35                 int temp = val;
36                 temp += i;
37                 val = (temp - 50);
38             }
39             checkInvariant(); // check invariant
40             System.out.println("WeakenedDerivedIncrementer value is: " + val);
41         }
42
43         private void checkInvariant(){
44             assert((0 <= val) && (val <= 50));
45         }
46     } // end WeakenedDerivedIncrementer class definition
47

```

Referring to example 24.5 — the `WeakenedDerivedIncrementer` class looks a lot like the `DerivedIncrementer` class with two notable exceptions. First, the precondition on the `increment()` method has been relaxed to allow a wider range of increment values. Second, the `if` statement that appears within the body of the `increment()` method starting on line 28 ensures the value of `i` used in the call to the base class version of `increment()` obeys its precondition. Example 24.6 shows the `WeakenedDerivedIncrementer` class being put through its paces in a modified version of the `MainTestApp` program.

24.6 *MainTestApp.java (mod 2)*

```

1      public class MainTestApp {
2          public static void main(String[] args){
3              Incrementer i1 = new Incrementer(0);
4              Incrementer i2 = new DerivedIncrementer(20);
5              Incrementer i3 = new WeakenedDerivedIncrementer(10);
6
7              i1.increment(1);
8              i1.increment(2);
9              i1.increment(3);
10             i1.increment(4);
11             i1.increment(5);
12             System.out.println("-----");
13             i2.increment(4);
14             i2.increment(5);
15             System.out.println("-----");
16             i3.increment(5);
17             i3.increment(6); // It does not throw exception here...
18             i3.increment(7); // nor here...
19             i3.increment(8); // nor here...
20             i3.increment(9); // nor here...
21             i3.increment(10); // nor here...

```



```

22         i3.increment(11); // ...but here it does!
23
24     } // end main() method
25 } // end MainTestApp clas definition

```

Referring to example 24.6 — a new `Incrementer` type reference named `i3` is declared and initialized to point to an object of type `WeakenedDerivedIncrementer`. Lines 16 through 22 call the `increment()` method via `i3`. As you can see, `WeakenedDerivedIncrementer`'s version of `increment()` allows a wider range of increment values. If steps weren't taken within the body of its `increment()` method to obey the contract of `Incrementer.increment()` then an exception would have been thrown on line 17.

However, from the point of view of a programmer whose expecting derived class objects to fulfill the contract of the base class `increment()` method, `WeakenedDerivedIncrementer` objects work just fine, as they allow the valid increment ranges of 1 through 5 which is what `Incrementer.increment()` methods expect. Figure 24-3 shows the results of running this modified version of `MainTestApp`.

```

Terminal — [csh]
[Rick-Millers-Computer:~/desktop/incrementer] swodag% java -ea MainTestApp
Incrementer object created with initial value of: 0
Incrementer object created with initial value of: 20
DerivedIncrementer object created with value: 20
Incrementer object created with initial value of: 10
WeakenedDerivedIncrementer object created with value: 10
Incrementer value is: 1
Incrementer value is: 3
Incrementer value is: 6
Incrementer value is: 10
Incrementer value is: 15
-----
Incrementer value is: 24
DerivedIncrementer value is: 24
Incrementer value is: 29
DerivedIncrementer value is: 29
-----
Incrementer value is: 15
WeakenedDerivedIncrementer value is: 15
WeakenedDerivedIncrementer value is: 21
WeakenedDerivedIncrementer value is: 28
WeakenedDerivedIncrementer value is: 36
WeakenedDerivedIncrementer value is: 45
WeakenedDerivedIncrementer value is: 5
Exception in thread "main" java.lang.AssertionError
    at WeakenedDerivedIncrementer.increment(WeakenedDerivedIncrementer.java:26)
    at MainTestApp.main(MainTestApp.java:22)
[Rick-Millers-Computer:~/desktop/incrementer] swodag%

```

Figure 24-3: Results of Running Example 24.6

STRENGTHENING PRECONDITIONS

So far you have seen how a derived class object can be substituted for an `Incrementer` class object when the derived class's `increment()` method adopts the same precondition or weakens the precondition of the `Incrementer` class's `increment()` method. When preconditions are kept the same or weakened in the overriding methods of a derived class, objects of the derived class type can be substituted for base class objects with little problem. However, if you happen to strengthen the precondition of an overriding derived class method you will break the code that relies on the original preconditions specified for the base class method.

A strengthening precondition in a derived class method places limits on or restricts the original precondition specified in the base class method it is overriding. In the case of `Incrementer` and its possible derived classes, the preconditions on a derived version of `increment()` can be strengthened to limit the range of authorized increment values to, say, 1 through 3. This would effectively break any code that relies on the `Incrementer`'s version of the `increment()` method that allows the increment values 1 through 5.

Example 24.7 gives the code for a class named `StrengthenedDerivedIncrementer` whose `increment()` method overrides the base class version and strengthens the precondition.

```

1     public class StrengthenedDerivedIncrementer extends Incrementer {
2         /*****
3             Class invariant: 0 <= val <= 50
4             *****/
5         private int val = 0;
6

```

24.7 *StrengthenedDerivedIncrementer.java*

```

7      /*****
8      Constructor Method: DerivedIncrementer(int i)
9          precondition: ((0 <= i) && (i <= 50))
10         postcondition: 0 <= val <= 50
11      *****/
12     public StrengthenedDerivedIncrementer(int i){
13         super(i);
14         assert((0 <= i) && (i <= 50)); // enforce precondition
15         val = i;
16         System.out.println("StrengthenedDerivedIncrementer object created with value: " + val);
17         checkInvariant();
18     }
19
20     /*****
21     Method: void increment(int i)
22         precondition: ((0 < i) && (i <= 3))
23         postcondition: 0 <= val <= 50
24     *****/
25     public void increment(int i){
26         assert((0 < i) && (i <= 3)); // enforce precondition
27         super.increment(i);
28         if((val+i) <= 50){
29             val += i;
30         }else{
31             int temp = val;
32             temp += i;
33             val = (temp - 50);
34         }
35         checkInvariant(); // check invariant
36         System.out.println("StrengthenedDerivedIncrementer value is: " + val);
37     }
38
39     private void checkInvariant(){
40         assert((0 <= val) && (val <= 50));
41     }
42
43     } // end StrengthenedDerivedIncrementer class definition

```

Referring to example 24.7 — the `StrengthenedDerivedIncrementer` class places a restriction on the original `increment()` method precondition by limiting the authorized increment values to 1 through 3. Example 24.8 shows the `StrengthenedDerivedIncrementer` class in action. Figure 24-4 shows the results of running this program.

24.8 *MainTestApp.java (mod 3)*

```

1     public class MainTestApp {
2         public static void main(String[] args){
3             Incrementer i1 = new Incrementer(0);
4             Incrementer i2 = new DerivedIncrementer(20);
5             Incrementer i3 = new WeakenedDerivedIncrementer(10);
6             Incrementer i4 = new StrengthenedDerivedIncrementer(10);
7
8             i1.increment(1);
9             i1.increment(2);
10            i1.increment(3);
11            i1.increment(4);
12            i1.increment(5);
13            System.out.println("-----");
14            i2.increment(4);
15            i2.increment(5);
16            System.out.println("-----");
17            i3.increment(5);
18            System.out.println("-----");
19            i4.increment(2); // OK so far...
20            i4.increment(3); // OK here too...
21            i4.increment(4); // Wait a minute...this should work!
22
23        } // end main() method
24    } // end MainTestApp clas definition

```

CHANGING THE POSTCONDITIONS OF DERIVED CLASS METHODS

Derived class method postconditions can be adopted, weakened, or strengthened just like preconditions. However, unlike preconditions, where a weakening condition is preferred to a strengthening condition, the opposite is true for postconditions: *A derived class method should specify and implement a stronger, rather than weaker, postcondition.*

```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/incrementer] swadog% java -ea MainTestApp
Incrementer object created with initial value of: 0
Incrementer object created with initial value of: 20
DerivedIncrementer object created with value: 20
Incrementer object created with initial value of: 10
WeakenedDerivedIncrementer object created with value: 10
Incrementer object created with initial value of: 10
StrengthenedDerivedIncrementer object created with value: 10
Incrementer value is: 1
Incrementer value is: 3
Incrementer value is: 6
Incrementer value is: 10
Incrementer value is: 15
-----
Incrementer value is: 24
DerivedIncrementer value is: 24
Incrementer value is: 29
DerivedIncrementer value is: 29
-----
Incrementer value is: 15
WeakenedDerivedIncrementer value is: 15
-----
Incrementer value is: 12
StrengthenedDerivedIncrementer value is: 12
Incrementer value is: 15
StrengthenedDerivedIncrementer value is: 15
Exception in thread "main" java.lang.AssertionError
    at StrengthenedDerivedIncrementer.increment(StrengthenedDerivedIncrementer.java:26)
    at MainTestApp.main(MainTestApp.java:21)
[Rick-Millers-Computer:~/desktop/incrementer] swadog%

```

Figure 24-4: Results of Running Example 24.8

The `Incrementer` and its derived class examples shown previously each had their own private attribute that was part of each class's invariant. (*Incrementer.val*, *DerivedIncrementer.val*, etc.) Each class's `increment()` method had a separate postcondition to preserve each class's invariant. The two postconditions did not conflict or contradict and were therefore compatible.

If, on the other hand, `Incrementer.val` had been declared `protected`, and was inherited and used by its derived classes, then derived versions of the `increment()` method would need a postcondition that either maintained the class invariant specified by the `Incrementer` class (*adopting postcondition*) or a postcondition that strengthened `Incrementer`'s class invariant (*strengthening postcondition*).

A weakening postcondition will cause problems. Consider for a moment what would happen if a derived class version of `increment()` allowed inherited `Incrementer.val` to assume values outside the range of those allowed by `Incrementer`'s class invariant specification. Disaster would strike the code sooner than later. (*Assuming some code somewhere depended upon `Incrementer` objects being within their specified, valid states.*)

SPECIAL CASES OF PRECONDITIONS AND POSTCONDITIONS

Method preconditions can specify and enforce more than just the values of method parameters, and postconditions can specify and enforce more than just class invariant states.

A method precondition can, for example, specify that the class invariant must hold true or that a combination of conditions hold true before it can do its job properly. A method postcondition can, in addition to enforcing the class invariant, specify the state of the object or reference the method returns (*if any*), or it can specify any number of conditions hold true upon completion of the method call. The conditions or combination of conditions imposed by derived class overriding method preconditions and postconditions can be weakening or strengthening.

The weakening and strengthening effects of preconditions and postconditions can apply to more than just simple conditions. Method parameter types and return types all play a part and are discussed below.

Method ARGUMENT Types

Derived class method preconditions can be weakened or strengthened by their method parameter types. An overriding method must agree with the method it overrides in the *type*, *number*, and *order* of its method parameters. Method parameter types can belong to a type hierarchy. This means that a method parameter might be related to another class via a subtype or supertype relationship.

A derived class method that declares a parameter whose type is a base class to the matching parameter declared by the base class's version of the method, then the derived class method is an overriding method. If, however, the derived class method declares a parameter that is a subclass of the parameter type declared by the base class method then the derived class method hides the base class's version of the method. This is due to the transitive nature of subtypes. (*i.e.*, *Given two types, Base and Derived, if Derived extends or implements Base, then Derived is a Base but a Base is not a Derived.*)

In other words, an overriding method can only provide a weakening precondition with regards to parameter types because to strengthen the parameter type required would result in the declaration of a new method, (*i.e.*, *method overloading*) (*requiring a new type from the point of view of the base class version of the method*) not the overriding of the base class method. To illustrate this point assume there exists the class inheritance hierarchy shown in figure 24-5.

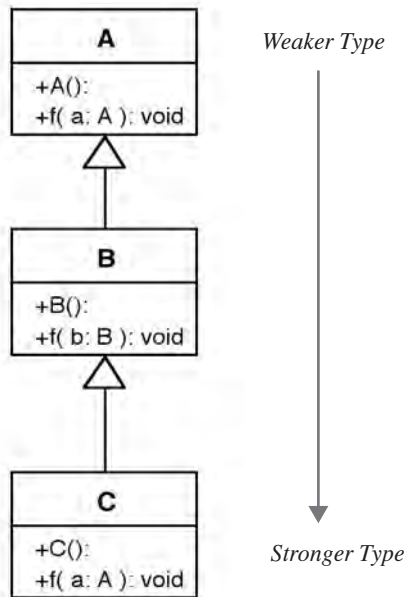


Figure 24-5: Strong vs. Weak Types

Each method `f()` in each class `A` and `C` requires a reference to an object of type `A`. Method `f()` in class `B` specifies a reference to an object of type `B`. Therefore, method `B.f()` is an overloading method while method `C.f()` is an overriding method. Examples 24.9 through 24.11 give the code for classes `A`, `B`, and `C`. Example 24.12 puts these classes through their paces and figure 24-6 shows the results of running this program.

```

1      public class A {
2          public A(){
3              System.out.println("A object created!");
4          }
5
6          public void f(A a){
7              System.out.println("A.f() called!");
8          }
9      }
  
```

24.9 A.java

```

1      public class B extends A {
2          public B(){
3              System.out.println("B object created!");
4          }
5
6          public void f(B b){
7              System.out.println("B.f() called!");
8          }
9      }
  
```

24.10 B.java

24.11 C.java

```

1 public class C extends B {
2     public C(){
3         System.out.println("C object created!");
4     }
5
6     public void f(A a){
7         System.out.println("C.f() called!");
8     }
9 }

```

24.12 MainTestApp.java

```

1 public class MainTestApp {
2     public static void main(String[] args){
3         A a1 = new A();
4         a1.f(new A()); // A's method called
5
6         System.out.println("-----");
7
8         A a2 = new B();
9         a2.f(new A()); // A's method called
10        a2.f(new B()); // A's method called
11
12        System.out.println("-----");
13
14        B b1 = new C();
15        b1.f(new A()); // C's overriding method called
16        b1.f(new B()); // B's overloaded method called
17        b1.f(new C()); // B's overloaded method called
18
19        System.out.println("-----");
20
21        A a3 = new C();
22        a3.f(new A()); // C's overriding method called
23        a3.f(new B()); // C's overriding method called
24        a3.f(new C()); // C's overriding method called
25
26    } // end main() method
27 } // end MainTestApp program

```

```

[Rick-Millers-Computer:~/desktop/strong_weak_types] swodog$ java MainTestApp
A object created!
A object created!
A.f() called!
-----
A object created!
B object created!
A object created!
A.f() called!
A object created!
B object created!
A.f() called!
-----
A object created!
B object created!
C object created!
A object created!
C.f() called!
A object created!
B object created!
B.f() called!
A object created!
B object created!
C object created!
B.f() called!
-----
A object created!
B object created!
C object created!
A object created!
C.f() called!
A object created!
B object created!
C.f() called!
A object created!
B object created!
C object created!
C.f() called!
[Rick-Millers-Computer:~/desktop/strong_weak_types] swodog$

```

Figure 24-6: Results of Running Example 24.12

Method Return Types

Method return types are considered special cases of postconditions. A reference to an object may be returned from a method as a result of its execution. Refer again to the inheritance hierarchy illustrated in figure 24-5. If a snippet of client code expects a return type from a method to be of a certain type, the method can strengthen that condition and return a subtype of the type expected. This strengthening of return types is in line with the strengthening usually required of postconditions.

THREE RULES OF THE SUBSTITUTION PRINCIPLE

In their book *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, Barbara Liskov and John Guttag say that the substitution principle must support three properties: the signature rule, the methods rule, and the properties rule. Each of these rules are discussed below.

SIGNATURE RULE

The signature rule deals with the methods published or made public by a type specification. In Java these methods would have public accessibility. For a subtype to obey the signature rule it must support all the methods published by its base class and that each overriding method is compatible with the method it overrides. Java enforces this type compatibility.

Methods Rule

The methods rule says that calls to overriding methods should behave like the base class methods they override. A type may be substitutable from a strictly type perspective but the behavior may be all wrong. Correct behavior of overriding methods is the aim of LSP and DbC.

PROPERTIES RULE

The properties rule is concerned with the preservation of provable base class properties by subtype behavior. A subtype should preserve the base class invariant. If a subtype's behavior violates a base class invariant then it is breaking the properties rule.

Quick Review

The preconditions of a derived class method should either adopt the same or weaker preconditions as the base class method it is overriding. A derived class method should never strengthen the preconditions specified in a base class version of the method. Derived class methods that strengthen base class method preconditions will render it impossible for programmers to reason about the behavior of subtype objects and lead to broken code should the ill-behaved derived class object be substituted for a base class object.

Method parameter types are considered special cases of preconditions. Preconditions should be weakened in the overriding Method, therefore, parameter types should be the same or weaker than the parameter types of the method being overridden. A base class is considered a weaker type than one of its subclasses.

Method return types are considered special cases of postconditions. The return type of an overriding method should be stronger than the type expected by the client code. A subclass is considered a stronger type than its base class.

THE OPEN-CLOSED PRINCIPLE

Software systems change over time. Change takes many forms, but changing and evolving system requirements provide the primary catalyst. A software system must accommodate change. It must evolve gracefully throughout its useful life cycle. A software system that is rigid, fragile, and change-resistant exhibits bad design. A software system

that is resilient, flexible, and extensible possesses the hallmark characteristics of a well-founded object-oriented architecture. The open-closed principle (OCP) provides the necessary framework for achieving an extensible and accommodating software architecture.

Formulated by Bertrand Meyer, the open-closed principle makes the following assertion:

*Software modules must be designed and implemented in a manner
that opens them for extension but closes them for modification.*

Said another way, changes to software modules should be avoided and new system functionality added by writing new code. It should be noted that writing code that is easy to extend and maintain is a requirement in and of itself. Writing such code takes longer initially but pays a big dividend later. I call it the design dividend.

Achieving The Open-Closed Principle

The key to writing code that conforms to the open-closed principle is to depend upon abstractions, not upon implementations. The reason — abstractions tend to be more stable. (*Correctly designed abstractions are very stable!*) This is achieved in Java through the use of abstract base classes or interfaces and dynamic polymorphic behavior. Code should rely only upon the interface methods and behavior promised via abstract methods. A code module that relies only upon abstractions will exhibit the characteristic of being closed to the need for modification yet open to the possibility of extension.

An OCP Example

A good example of code written with the OCP in mind given in examples 24.13 through 24.21. This code implements a simple naval fleet model where vessels of various types can be constructed with different types of power plants and weapons. Figure 24-7 gives the UML diagram for the naval fleet class inheritance hierarchy. Example 24.22 offers a short program showing the naval fleet classes in action, and figure 24-8 shows the results of running this program.

24.13 Vessel.java

```

1      public abstract class Vessel {
2          private Plant its_plant = null;
3          private Weapon its_weapon = null;
4          private String its_name = null;
5
6          public Vessel(Plant plant, Weapon weapon, String name){
7              its_weapon = weapon;
8              its_plant = plant;
9              its_name = name;
10             System.out.println("The vessel " + its_name + " created!");
11         }
12
13         /* *****
14            Public Abstract Methods - must be implemented in
15            derived classes.
16            *****/
17         public abstract void lightoffPlant();
18         public abstract void shutdownPlant();
19         public abstract void trainWeapon();
20         public abstract void fireWeapon();
21
22         /* *****
23            toString() Method - may be overridden in subclasses.
24            *****/
25         public String toString(){
26             return "Vessel name: " + its_name + " " + its_plant.toString() +
27                 " " + its_weapon.toString();
28         }
29         protected Weapon getWeapon(){ return its_weapon; }
30         protected Plant getPlant(){ return its_plant; }
31
32     } // end Vessel class definition

```

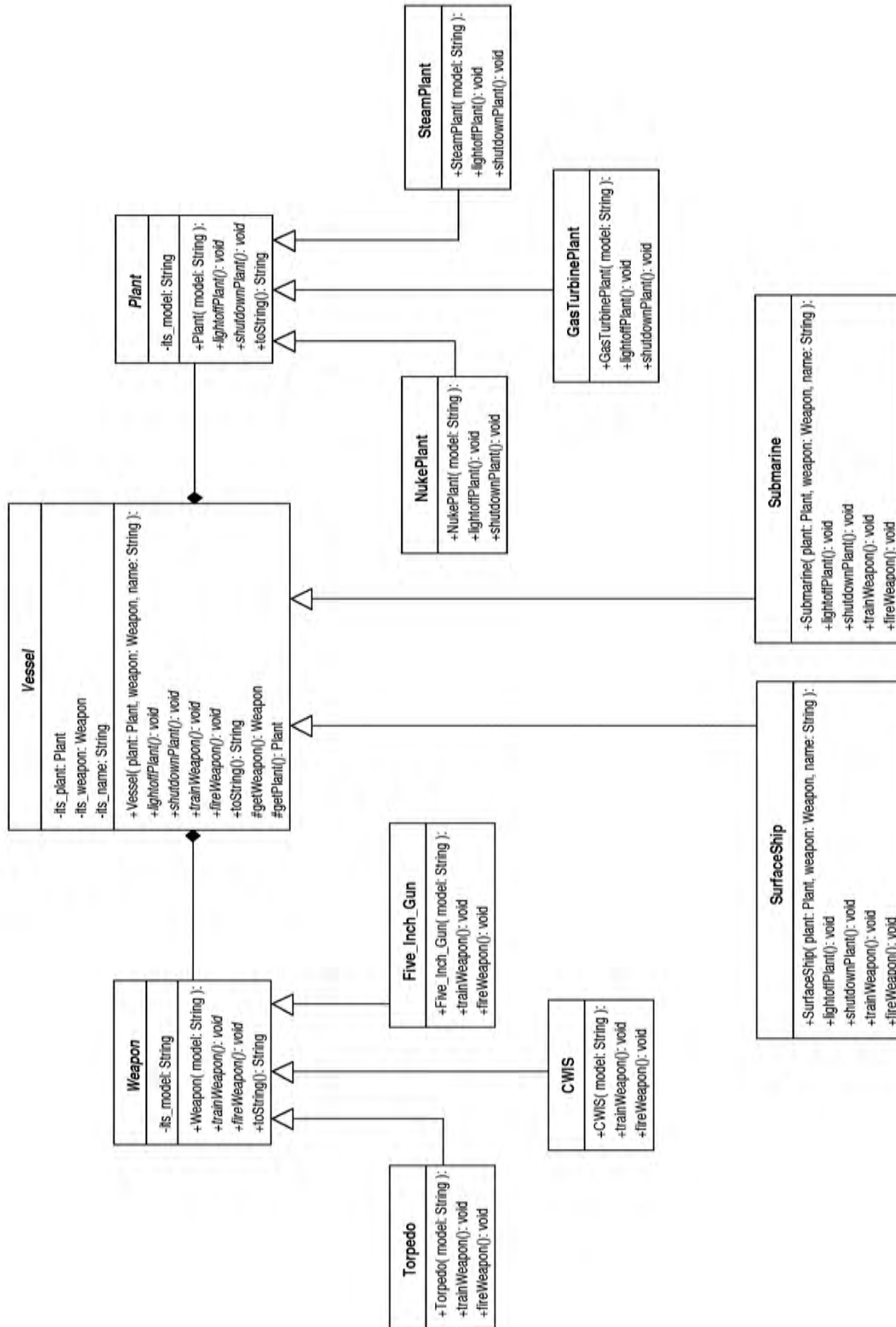


Figure 24-7: Naval Fleet Class Inheritance Hierarchy

24.14 *Plant.java*

```

1      public abstract class Plant {
2          private String its_model = null;
3          public Plant(String model){
4              its_model = model;
5          }
6          public abstract void lightoffPlant();
7          public abstract void shutdownPlant();
8
9          public String toString(){ return "Plant model: " + its_model; }
10     }

```

24.15 *Weapon.java*

```

1      public abstract class Weapon {
2          private String its_model = null;
3
4          public Weapon(String model){
5              its_model = model;
6              System.out.println("Weapon object created!");
7          }
8
9          public abstract void trainWeapon();
10         public abstract void fireWeapon();
11
12         public String toString(){ return "Weapon model: " + its_model; }
13     }

```

24.16 *CIWS.java*

```

1      public class CIWS extends Weapon {
2
3          public CIWS(String model){
4              super(model);
5              System.out.println("CIWS object created!");
6          }
7
8          public void trainWeapon(){
9              System.out.println("CIWS is locked on target!");
10         }
11
12         public void fireWeapon(){
13             System.out.println("The CIWS roars to life and fires a zillion bullets at the target!");
14         }
15     }

```

24.17 *Torpedo.java*

```

1      public class Torpedo extends Weapon {
2
3          public Torpedo(String model){
4              super(model);
5              System.out.println("Torpedo object created!");
6          }
7
8          public void trainWeapon(){
9              System.out.println("Torpedo is locked on target!");
10         }
11
12         public void fireWeapon(){
13             System.out.println("Fish in the water, heading towards target!");
14         }
15     }

```

24.18 *Five_Inch_Gun.java*

```

1      public class Five_Inch_Gun extends Weapon {
2
3          public Five_Inch_Gun(String model){
4              super(model);
5              System.out.println("Five Inch Gun object created!");
6          }
7
8          public void trainWeapon(){
9              System.out.println("Five Inch Gun is locked on target!");
10         }
11
12         public void fireWeapon(){
13             System.out.println("Blam! Blam! Blam!");
14         }
15     }

```

24.19 *SteamPlant.java*

```

1      public class SteamPlant extends Plant {
2
3          public SteamPlant(String model){
4              super(model);
5              System.out.println("SteamPlant object created!");
6          }
7
8          public void lightoffPlant(){
9              System.out.println("Steam pressure is rising!");
10         }
11
12         public void shutdownPlant(){
13             System.out.println("Steam plant is secure!");
14         }
15     }

```

24.20 *NukePlant.java*

```

1      public class NukePlant extends Plant {
2
3          public NukePlant(String model){
4              super(model);
5              System.out.println("NukePlant object created!");
6          }
7
8          public void lightoffPlant(){
9              System.out.println("Nuke plant is critical!");
10         }
11
12         public void shutdownPlant(){
13             System.out.println("Nuke plant is secure!");
14         }
15     }

```

24.21 *GasTurbinePlant.java*

```

1      public class GasTurbinePlant extends Plant {
2
3          public GasTurbinePlant(String model){
4              super(model);
5              System.out.println("GasTurbinePlant object created!");
6          }
7
8          public void lightoffPlant(){
9              System.out.println("Gas Turbine is running and ready to go!");
10         }
11
12         public void shutdownPlant(){
13             System.out.println("Gas Turbine is secure!");
14         }
15     }

```

24.22 *FleetTestApp.java*

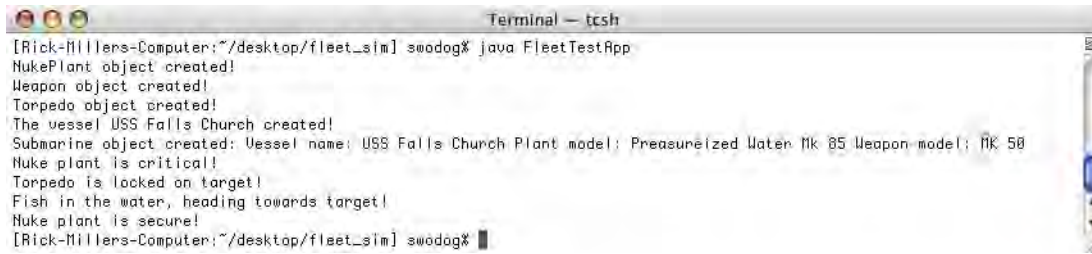
```

1      public class FleetTestApp {
2          public static void main(String[] args){
3              Vessel v1 = new Submarine(new NukePlant("Preasureized Water Mk 85"), new Torpedo("MK 50"),
4                                      "USS Falls Church");
5
6              v1.lightoffPlant();
7              v1.trainWeapon();
8              v1.fireWeapon();
9              v1.shutdownPlant();
10         }
11     }
12 } // end FleetTestApp class definition

```

Quick Review

The open-closed principle (OCP) attempts to optimize object-oriented software architecture design so it can accommodate change. Software modules should be designed so they are closed to modification yet open to extension. The OCP is achieved by depending upon software abstractions. In Java this means designing with abstract base classes or interfaces while keeping the goal of dynamic polymorphic behavior in mind. The OCP relies heavily upon the Liskov substitution principle and design by contract (LSP/DbC).



```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/fleet_sim] swodog% java FleetTestApp
NukePlant object created!
Weapon object created!
Torpedo object created!
The vessel USS Falls Church created!
Submarine object created: Vessel name: USS Falls Church Plant model: Pressurized Water Mk 85 Weapon model: MK 58
Nuke plant is critical!
Torpedo is locked on target!
Fish in the water, heading towards target!
Nuke plant is secure!
[Rick-Millers-Computer:~/desktop/fleet_sim] swodog%

```

Figure 24-8: Results of Running Example 24.22

THE DEPENDENCY INVERSION PRINCIPLE

When used together in a disciplined approach, the OCP and the LSP/DbC yield a desirable inversion of program module dependencies that is different from the usual top-down module dependencies attained with functional decomposition. This dependency inversion is generalized into a principle in its own right known as the Dependency Inversion Principle (DIP). Robert C. Martin stated the definition of the DIP in two parts that I've paraphrased here:

*A. High-level modules should not depend upon low-level modules.
Both should depend upon abstractions.*

*B. Abstractions should not depend upon details. Details should
depend upon abstractions.*

CHARACTERISTICS OF BAD SOFTWARE ARCHITECTURE

When a software module depends on the details of a lower-level software module it is hard to change and hard to reuse. Consider the software module hierarchy shown in figure 24-9 where high-level modules depend on low-level modules.

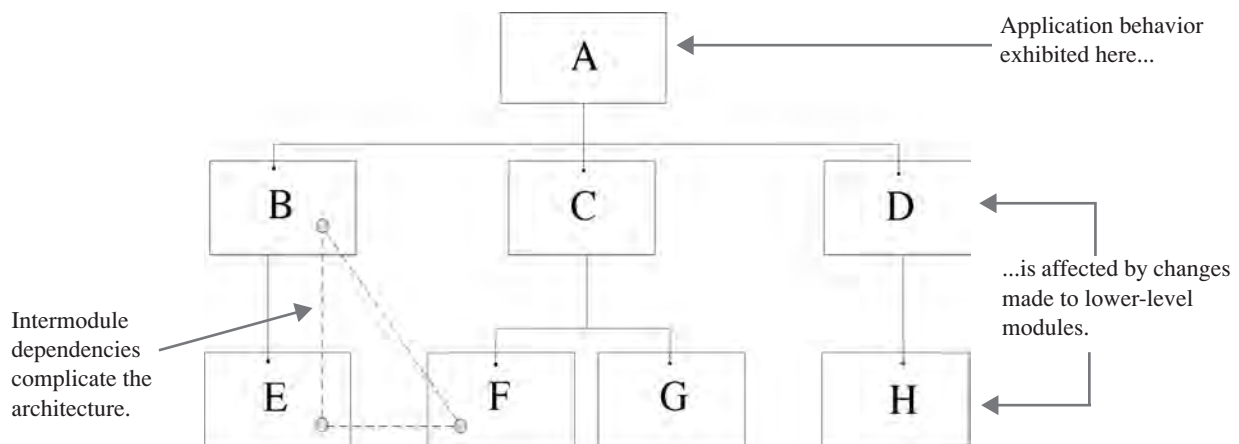


Figure 24-9: Traditional Top-Down Functional Dependencies

The behavior of module A depends on modules B, C, and D. The behavior of module B depends on module E, module C depends on the behavior of modules F and G, and module D depends on module H. A change to module E affects module B which, in turn, affects module A. Any intermodule dependencies, such as global variables, will further complicate the issue. A complex software system sporting this sort of architecture will have the undesirable characteristics of bad design, namely, it will be *fragile*, *rigid*, and *immobile*.

A fragile software architecture is one that breaks in unexpected ways when a change is made to one or more software modules. Fragile software leads to rigid software.

A rigid software architecture is one that is so difficult and painful to change that programmers do not want to change it.

An immobile software architecture is characterized by the inability to successfully extract the software module for reuse in another system. The software module may exhibit desirable behavior but if it is too dependent on other modules or anchored to the application architecture by intermodule dependencies then it will be difficult if not impossible to reuse in another similar context. If it is easier to rewrite a module from scratch than it is to adopt and reuse the module then the module is immobile.

CHARACTERISTICS OF GOOD SOFTWARE ARCHITECTURE

Object-oriented software architectures that subscribe to the OCP and the LSP/DbC will depend heavily upon abstractions. These abstractions will appear at or near the top of the software module hierarchy. Refer again to the naval fleet class hierarchy shown in figure 24-7. The Vessel, Weapon, and Plant abstract base classes serve as the foundation for all behavior inherited by the lower-level implementation classes. This inheritance relationship means that the lower-level derived classes are dependent upon the behavior specified by the higher-level base class abstractions.

The key to success with the DIP lies in choosing the right software abstractions. A software architecture based upon the right kinds of abstractions will exhibit the desirable characteristic of being easy to extend. It will be flexible because of its extensibility, it will be non-rigid in that the addition of new functionality via new derived classes will not affect the behavior of existing abstractions. Lastly, software modules that depend upon abstractions can generally be reused in a wider variety of contexts, thus achieving a greater degree of mobility.

SELECTING THE RIGHT ABSTRACTIONS TAKES EXPERIENCE

The ability to identify essential software component abstractions takes practice and experience. However, applying the OCP and the LSP/DbC in your object-oriented software architecture design will yield a better design, even if you do not get all the abstractions right the first time around.

Quick Review

The OCP and the LSP/DbC, when applied together, result in the realization of a third design principle known as the Dependency Inversion Principle (DIP). The key to the DIP is that high-level software modules should not rely on low-level details, and that software modules at all hierarchy levels should rely upon abstractions. When a software architecture achieves the goals of the DIP it is easier to extend and maintain (*i.e., it is flexible and non-rigid*). Software modules that conform to the DIP are easier to reuse in other contexts (*i.e., they are mobile*).

TERMS AND DEFINITIONS

The terms and definitions listed in table 24-1 were used throughout this chapter:

Term	Definition
<i>Abstraction</i>	The separation of the important from the unimportant. (i.e. interface vs. implementation)
<i>Abstract Data Type</i>	A type specification that separates the interface to the type from the type's implementation. An abstract data type represents a set of objects that can be manipulated via a set of interface methods.
<i>Supertype</i>	An abstract data type that serves as a specification for related subtypes.

Table 24-1: Terms and Definitions Related to the LSP

Term	Definition
<i>Subtype</i>	An abstract data type that derives all or part of its specification from another abstract data type. A subtype can inherit the specification of a supertype then add something extra if required.
<i>Type Specification</i>	A declaration of the behavioral properties of an abstract data type. A specification describes the important characteristics of the data abstraction.
<i>Encapsulation</i>	The act of hiding private implementation details behind a publicly accessible interface.
<i>Precondition</i>	A condition, constraint, or set of constraints that must hold true during a call to an abstract data type interface method to ensure its proper operation.
<i>Postcondition</i>	A condition, constraint, or set of constraints that must be satisfied when an abstract data type method completes execution.
<i>Inheritance Hierarchy</i>	A set of abstract data type specifications that implement a supertype and subtype relationship between each abstract data type.
<i>Class</i>	The declaration of an abstract data type specifying a set of attributes and interface methods common to a set of objects.
<i>Abstract Class</i>	The declaration of an abstract data type specifying a set of attributes and interface methods common to a set of objects. One or more interface methods are declared to be abstract and are therefore deferred to subclasses for implementation.
<i>Subclass</i>	A declaration of an abstract data type taking all or part of its specification from another, possibly abstract, class.
<i>Class Invariant</i>	An assertion about the state of an object which must hold true for all possible states the object may assume.

Table 24-1: Terms and Definitions Related to the LSP

SUMMARY

Well-designed software architectures exhibit three characteristics: 1) they are easy to understand, 2) they are easy to reason about, and 3) they are easy to extend.

The Liskov Substitution Principle (LSP) and Bertrand Meyer's Design by Contract (DbC) programming are closely related principles designed to enable programmers to better reason about subtype behavior.

The preconditions of a derived class method should either adopt the same or weaker preconditions as the base class method it is overriding. A derived class method should never strengthen the preconditions specified in a base class version of the method. Derived class methods that strengthen base class method preconditions will render it impossible for programmers to reason about the behavior of subtype objects and lead to broken code should the ill-behaved derived class object be substituted for a base class object.

Method parameter types are considered special cases of preconditions. Preconditions should be weakened in the overriding Method, therefore, parameter types should be the same or weaker than the parameter types of the method being overridden. A base class is considered a weaker type than one of its subclasses.

Method return types are considered special cases of postconditions. The return type of an overriding method should be stronger than the type expected by the client code. A subclass is considered a stronger type than its base class.

The open-closed principle (OCP) attempts to optimize object-oriented software architecture design so it can accommodate change. Software modules should be designed so they are closed to modification yet open to extension. The OCP is achieved by depending upon software abstractions. In Java this means designing with abstract base classes or interfaces while keeping the goal of dynamic polymorphic behavior in mind. The OCP relies heavily upon the Liskov substitution principle and design by contract (LSP/DbC).

The OCP and the LSP/DbC, when applied together, result in the realization of a third design principle known as the Dependency Inversion Principle (DIP). The key to the DIP is that high-level software modules should not rely on low-level details, and that software modules at all hierarchy levels should rely upon abstractions. When a software architecture achieves the goals of the DIP it is easier to extend and maintain (*i.e., it is flexible and non-rigid*). Software modules that conform to the DIP are easier to reuse in other contexts (*i.e., they are mobile*).

Skill-Building Exercises

1. **Research:** Procure a copy of Bertrand Meyer's excellent book *Object-Oriented Software Construction, Second Edition*, and read it from front to back.
2. **Research:** Procure a copy of Robert C. Martin's book *Designing Object-Oriented C++ Applications Using The Booch Method*. (Although the Booch diagramming notation has been superseded by the Unified Modeling Language, and the code examples are given in C++, the Java student will still gain much from reading this excellent work.)
3. **Research:** Conduct a web search for the keywords Liskov substitution principle, open-closed principle, dependency inversion principle, and Meyer design by contract programming.
4. **Design Modification:** Create a new checked exception named `InvalidPreconditionException` and utilize it instead of the Java assertion mechanism to signal invalid preconditions. (*Hint: The `Incrementer.incrementer()` method, and any methods that override it, should throw this exception if the precondition is invalid.*)

Suggested Projects

1. **RobotRat:** Evaluate your latest version of `RobotRat` and apply each of the three design principles to your design. What improvements, if any, can be realized by applying each principle? How would your design have to be modified to take full advantage of each of the three design principles?

Self-Test Questions

1. List and describe the preferred characteristics of an object-oriented architecture.
2. State the definition of the Liskov substitution principle.
3. Define the term class invariant.
4. What is the purpose of a method precondition?
5. What is the purpose of a method postcondition?
6. List and describe the three rules of the substitution principle.
7. Write the definition and goals of the open-closed principle.
8. Explain how the open-closed principle uses the Liskov substitution principle and Meyer design by contract programming to achieve its goals.

9. Write the definition and goals of the dependency inversion principle.
10. Explain how the dependency inversion principle builds upon the open-closed principle and the Liskov substitution principle/Meyer design by contract programming.

REFERENCES

Barbara Liskov, *Data Abstraction and Hierarchy*, SIGPLAN Notices, 23,5 (May 1988).

W. Al-Ahmad, *On The Interaction of Programming By Contract and Liskov Substitution Principle*.

Bertrand Meyer, *Applying "Design by Contract"*, IEEE Computer, Vol. 25 Number 10, October 1992, pp. 40 - 51.

Barbara H. Liskov, Jeannette M. Wing, *A Behavioral Notion of Subtyping*. ACM Transactions on Programming Languages and Systems, Vol 16, No 6, November 1994, pp. 1811-1841.

James O. Coplien, *Advanced C++: Programming Styles and Idioms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1992. ISBN: 0-201-54855-0

Barbara Liskov, John Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Boston, Massachusetts, 2001. ISBN: 0-201-65768-6

Robert C. Martin. *Designing Object-Oriented C++ Applications Using The Booch Method*. Prentice Hall, Englewood Cliffs, New Jersey, 1995. ISBN: 0-13-203837-4

Bertrand Meyer. *Towards practical proofs of class correctness*, to appear in Proc. 3rd International B and Z Users Conference (ZB 2003), Turku (Finland), June 2003, ed. Didier Bert, Springer-Verlag, 2003.

Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall PTR, Upper Saddle River, New Jersey 07458. ISBN: 0-13-629155-4

Rick Miller. *C++ For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-02-8

NOTES

CHAPTER 25



Dried Shrub

Helpful Design PATTERNS

LEARNING OBJECTIVES

- *STATE THE PURPOSE AND USE OF DESIGN PATTERNS*
- *EXPLAIN WHY DESIGN PATTERNS ARE A FORM OF KNOWLEDGE REUSE*
- *DESCRIBE THE PURPOSE OF THE FACTORY PATTERN*
- *DESCRIBE HOW TO CREATE A FACTORY CLASS THAT LOADS OTHER CLASSES BY CLASS NAME*
- *DESCRIBE THE PURPOSE OF THE SINGLETON PATTERN*
- *DESCRIBE THE PURPOSE OF THE COMMAND PATTERN*
- *DESCRIBE THE PURPOSE OF THE MODEL-VIEW-CONTROLLER (MVC) PATTERN*
- *DESCRIBE HOW TO DYNAMICALLY LOAD A CLASS INTO THE JVM BY NAME USING THE CLASS.forName() METHOD*
- *COMBINE THE MVC, FACTORY, SINGLETON, AND COMMAND PATTERNS TO FORMULATE FLEXIBLE APPLICATION ARCHITECTURES*
- *DEMONSTRATE YOUR ABILITY TO UTILIZE DESIGN PATTERNS IN YOUR PROGRAMS*

INTRODUCTION

This chapter offers a brief introduction to the topic of software design patterns. Throughout this book you informally encountered several design patterns including the Singleton, Factory, and the Façade. Here I will explain the meaning of the term design pattern, why they are considered to be a form of knowledge reuse, and how they can help you write better software.

Too many design patterns exist to offer them complete treatment in the limited space of this chapter. However, I feel it is important for you to at least understand how design patterns came to be and to learn a few of them to keep in your back pocket for use on your next project. To this end I will focus the discussion in this chapter on the purpose and use of the Singleton, Factory, Model-View-Controller, and Command patterns. I will show you how to combine these patterns to create robust, flexible application architectures. I will also show you how to separate business logic from presentation logic with the help of the Model-View-Controller pattern. Along the way you will also learn how to process application commands polymorphically using the Command pattern.

An understanding of design patterns will forever change the way you approach the task of building software architectures.

SOFTWARE DESIGN PATTERNS AND HOW THEY CAME TO BE

Each new advance in the field of software engineering brings with it the promise to build better software. The positive impact made upon the software engineering profession by object-oriented analysis, design, and programming techniques cannot be denied. However, insufficient training and experience can, and usually does, result in poorly-designed systems that are impossible to maintain. How can you then, if you are a novice, best capitalize on and apply the lessons-learned by software developers who have come before you? The answer is — learn how to use software design patterns to build your application architectures.

WHAT EXACTLY IS A SOFTWARE DESIGN PATTERN?

A software design pattern is a form of knowledge reuse. Many extremely bright, talented software professionals working hard over the years to produce robust, reliable, flexible software architectures, noticed that for similar design problems they created similar design solutions. These similar design solutions consisted of a set of one or more related classes and object interactions. The real intellectual leap came when these engineers realized they could extract the essence of each design solution into a more general solution specification. These general solution specifications could then be readily reused and applied as the architectural basis of specific design solutions for the design problems they addressed. These general design specifications are referred to as *software design patterns*. When you apply software design patterns in your application architecture you are standing upon the backs of giants.

ORIGINS

The term design pattern is borrowed from the work of an architect named Christopher Alexander. Alexander is not a software architect; he is a building architect, and his work has had a tremendous philosophical influence upon the software design patterns movement. In his book *The Timeless Way Of Building*, Alexander lists three things necessary to build good buildings and towns: 1) the *timeless way*, 2) the *quality without a name* (QWAN), and 2) the *gate*. The objective is to create a building that manifests the quality without a name. The QWAN is something you immediately recognize when experienced yet is impossible to describe exactly. The QWAN can be achieved by building using the timeless way. The timeless way is the process by which living buildings and towns are created by implementing, or in Alexander's words, unfolding, one architectural design pattern at a time. Success in the application of the way depends upon the intensity of each pattern's implementation. The gate is a pattern language of architectural features that have been proven through the ages to work well for certain applications. *To achieve the QWAN you must pass through the gate in order to practice the timeless way.*

PATTERN SPECIFICATION

A pattern language manifests itself as a catalog of design patterns. In their book *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (referred to in the software community as the *Gang-of-Four*) describe each design pattern according to the following template:

Section Name	Comments
<i>Pattern Name and Classification</i>	The name of the pattern.
<i>Intent</i>	A description of the pattern's purpose and use.
<i>Also Known As</i>	Other names the pattern may be known by.
<i>Motivation</i>	The problem the pattern is trying to solve.
<i>Applicability</i>	Under what situations the pattern should be applied.
<i>Structure</i>	A graphical representation of the pattern in a notational language like UML.
<i>Participants</i>	The pattern's classes and/or objects and their responsibilities.
<i>Collaborations</i>	How a pattern's participants execute their responsibilities.
<i>Consequences</i>	The ramifications of the pattern's use.
<i>Implementation</i>	Advice on using the pattern.
<i>Sample Code</i>	Concrete pattern implementation example in a programming language like Java.
<i>Known Uses</i>	Examples of the pattern's application in the real world.
<i>Related Patterns</i>	Closely-related design patterns.

Table 25-1: Pattern Specification Template

I will not completely describe the few patterns I discuss in this chapter according to the template shown in table 25-1. I do, however, urge you to refer to the references listed at the end of the chapter to learn more about software design patterns and how you can use them to build better software.

Applying Software Design Patterns

Some software design patterns can be used stand-alone while others are meant to be combined with other patterns to form a complete solution. The best way to learn about design patterns is to review a design pattern catalog, such as that given in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by the Gang-of-Four, to get a feel for the different types of patterns and their application.

You don't have to be a pattern wizard to realize the benefits of using patterns in your programs. The rest of this chapter is devoted to showing you how four design patterns can be used to produce a robust, flexible application architecture.

Quick Review

Software design patterns are a form of knowledge reuse. Design patterns are general software architectural solutions to general software architectural problems. A design pattern serves as the basis for a specific solution implementation. A complete design pattern specification includes more than just a graphical representation. Some design patterns can be applied alone while others are meant to be combined with other design patterns.

THE SINGLETON PATTERN

The Singleton is used when your application needs only one instance, or a controlled number of instances, of a particular type. Examples of Singletons include application session objects and application properties objects. The Singleton is often used to implement other patterns such as the Factory. (*The Factory pattern is discussed below.*)

The general approach to implementing a Singleton is to create a class that has a protected or private constructor and a public static method named getInstance(). Example 25.1 shows the code for a class named CommandProperties that implements the Singleton design pattern.

25.1 CommandProperties.java

```

1      package com.pulpfreepress.utils;
2
3      import java.util.*;
4      import java.io.*;
5
6      public class CommandProperties extends Properties {
7
8          // class constants - default key strings
9          public static final String PROPERTIES_FILE           = "PROPERTIES_FILE";
10         public static final String NEWHOURLYEMPLOYEE_COMMAND = "NewHourlyEmployee";
11         public static final String NEWSALARIEDEMPLOYEE_COMMAND = "NewSalariedEmployee";
12         public static final String EXIT_COMMAND              = "Exit";
13         public static final String LIST_COMMAND               = "List";
14         public static final String SORT_COMMAND               = "Sort";
15         public static final String SAVE_COMMAND               = "Save";
16         public static final String EDITEMPLOYEE_COMMAND       = "EditEmployee";
17         public static final String DELETEEMPLOYEE_COMMAND     = "DeleteEmployee";
18         public static final String LOAD_COMMAND               = "Load";
19
20         // class constants - default value strings
21         private static final String PROPERTIES_FILE_VALUE= "Command.properties";
22         private static final String NEWHOURLYEMPLOYEE_COMMAND_CLASSNAME
23             = "com.pulpfreepress.commands.NewHourlyEmployeeCommand";
24         private static final String NEWSALARIEDEMPLOYEE_COMMAND_CLASSNAME
25             = "com.pulpfreepress.commands.NewSalariedEmployeeCommand";
26         private static final String EXIT_COMMAND_CLASSNAME
27             = "com.pulpfreepress.commands.ApplicationExitCommand";
28         private static final String LIST_COMMAND_CLASSNAME
29             = "com.pulpfreepress.commands.ListEmployeesCommand";
30         private static final String SORT_COMMAND_CLASSNAME
31             = "com.pulpfreepress.commands.SortEmployeesCommand";
32         private static final String SAVE_COMMAND_CLASSNAME
33             = "com.pulpfreepress.commands.SaveEmployeesCommand";
34         private static final String EDITEMPLOYEE_COMMAND_CLASSNAME
35             = "com.pulpfreepress.commands.EditEmployeeCommand";
36         private static final String DELETEEMPLOYEE_COMMAND_CLASSNAME
37             = "com.pulpfreepress.commands.DeleteEmployeeCommand";
38         private static final String LOAD_COMMAND_CLASSNAME
39             = "com.pulpfreepress.commands.LoadEmployeesCommand";
40
41         // class variables
42         private static CommandProperties _properties_object = null;
43
44         private CommandProperties( String properties_file ){
45             try{
46                 FileInputStream fis = new FileInputStream(properties_file);
47                 load(fis);
48             }catch(Exception e) {
49                 System.out.println("Problem opening properties file!");
50                 System.out.println("Bootstrapping properties...");
51                 try{
52                     FileOutputStream fos = new FileOutputStream(PROPERTIES_FILE_VALUE);
53                     setProperty(PROPERTIES_FILE, PROPERTIES_FILE_VALUE);
54                     setProperty(NEWHOURLYEMPLOYEE_COMMAND, NEWHOURLYEMPLOYEE_COMMAND_CLASSNAME);
55                     setProperty(NEWSALARIEDEMPLOYEE_COMMAND, NEWSALARIEDEMPLOYEE_COMMAND_CLASSNAME);
56                     setProperty(EXIT_COMMAND, EXIT_COMMAND_CLASSNAME);
57                     setProperty(LIST_COMMAND, LIST_COMMAND_CLASSNAME);
58                     setProperty(SORT_COMMAND, SORT_COMMAND_CLASSNAME);
59                     setProperty(SAVE_COMMAND, SAVE_COMMAND_CLASSNAME);
60                     setProperty(EDITEMPLOYEE_COMMAND, EDITEMPLOYEE_COMMAND_CLASSNAME);
61                     setProperty(DELETEEMPLOYEE_COMMAND, DELETEEMPLOYEE_COMMAND_CLASSNAME);
62                     setProperty(LOAD_COMMAND, LOAD_COMMAND_CLASSNAME);
63
64                     super.store(fos, "CommandProperties File - Edit Carefully");
65                     fos.close();

```

```

66         }catch(Exception e2){ System.out.println("Uh ohh...Bigger problems exist!"); }
67     }
68 }
69
70 /*****
71 * Private default constructor. Applications will get an instance via the getInstance() method.
72 * @see getInstance()
73 *****/
74 private CommandProperties(){
75     this(PROPERTIES_FILE_VALUE);
76 }
77
78 /*****
79 * The store() method attempts to persist its properties collection.
80 *****/
81 public void store(){
82     try{
83         FileOutputStream fos = new
84             FileOutputStream(getProperty(PROPERTIES_FILE));
85         super.store(fos, "CommandProperties File");
86         fos.close();
87     }catch(Exception e){ System.out.println("Trouble storing properties!"); }
88 }
89
90 /*****
91 * getInstance() returns a singleton instance if the CommandProperties object.
92 *****/
93 public static CommandProperties getInstance(){
94     if(_properties_object == null){
95         _properties_object = new CommandProperties();
96     }
97     return _properties_object;
98 }
99 } // end CommandProperties class definition

```

Referring to example 25.1 — the `CommandProperties` class is used to create a menu command-to-command handler class mapping file named *command.properties*. The `CommandProperties` class is used in the comprehensive example presented at the end of the chapter. Notice the declaration on line 42 of a private static field of type `CommandProperties` named `_properties_object`. Notice also that both constructors are declared private. The static `getInstance()` method begins on line 93. It ensures the existence of only one instance of `CommandProperties`.

Quick Review

The Singleton pattern is used when only one instance of a particular class type is required to exist in your program. The general approach to creating a Singleton is to make the constructor protected or private and provide a public static method named `getInstance()` that returns the same instance of the class in question.

THE FACTORY PATTERN

The Factory pattern is used to create classes whose purpose is to create and return objects or references to objects. Two well known Factory patterns include the *Abstract Factory* and the *Factory Method*. You've seen the Abstract Factory pattern in action in chapter 22 with the creation of the `IEmployeeFactory` interface and the `EmployeeFactory` class. Factory implementations are often Singletons as well.

Let's take a look once more at the `IEmployeeFactory` interface shown in example 25.2.

25.2 *IEmployeeFactory.java*

```

1     package com.pulpfreepress.model;
2
3     public interface IEmployeeFactory {
4         IEmployee getNewSalariedEmployee(String f_name, String m_name, String l_name, int dob_year,
5             int dob_month, int dob_day, String gender, String employee_number);
6         IEmployee getNewHourlyEmployee(String f_name, String m_name, String l_name, int dob_year,
7             int dob_month, int dob_day, String gender, String employee_number);
8     }

```

Each `IEmployeeFactory` method returns a reference to an object of type `IEmployee`, which in this case is either a `SalariedEmployee` or an `HourlyEmployee`. Each of `IEmployee`'s methods are an example of a Factory Method.

A limitation of `IEmployeeFactory` and any classes that implement this interface is that the factory methods, by their very name, place a conceptual limit on the type of objects that will be created. Two different implementations of the `getNewSalariedEmployee()` method may, in two different concrete implementations of `IEmployeeFactory`, create different flavors of salaried employees. However, from a polymorphic standpoint, it should make no difference, so long as whatever comes our way implements the `IEmployee` interface. Still, the method names place a conceptual limit on the sort of `IEmployee` objects we expect from an `IEmployeeFactory`.

THE DYNAMIC FACTORY

In many programming situations it would be nice to simply name the type of object we need, send that name to a factory, and have the factory make us an object of that type. You can do this with the Dynamic Factory. The Dynamic Factory combines the Factory pattern with dynamic class loading to achieve a flexible mechanism for object creation. The general approach to creating a Dynamic Factory is to create a class that contains a public method that takes a `String` argument representing the class name of the type of object you need to create. The method will then try and dynamically load the class into the JVM using the `Class.forName()` method and, if successful, create an object of that type and return its reference. The objects a Dynamic Factory creates must have default (*i.e.*, *no argument*) constructors. Example 25.3 gives a simple example of a Dynamic Factory named `InterfaceTypeFactory`.

25.3 *InterfaceTypeFactory.java*

```

1      public class InterfaceTypeFactory {
2
3          public static InterfaceType newObjectByClassName(String classname) {
4              Object o = null;
5              try{
6                  Class c = Class.forName(classname);
7                  o = c.newInstance();
8              }catch(Exception e){
9                  System.out.println("Problem loading class or creating instance!");
10             }
11             return (InterfaceType)o;
12         }
13     } // end InterfaceTypeFactory class definition

```

Referring to example 25.3 — the `InterfaceTypeFactory` class has one public static method named `newObjectByClassName(String classname)`. It takes a `String` argument representing the fully-qualified class name of the object to be created and returns a reference of type `InterfaceType`. For the method to work the class requested must exist and be in the system's classpath.

The following examples give the code for the `InterfaceType` interface and several classes that implement the interface. These are followed by a short program showing the `InterfaceTypeFactory` class in action. Figure 25-1 shows the results of running the `MainTestApp` program.

25.4 *InterfaceType.java*

```

1      public interface InterfaceType {
2          public String getMessage();
3      }

```

25.5 *ClassA.java*

```

1      public class ClassA implements InterfaceType {
2          private String message = "ClassA's message";
3          public String getMessage(){ return message; }
4      }

```

25.6 *ClassB.java*

```

1      public class ClassB implements InterfaceType {
2          private String message = "ClassB's message";
3          public String getMessage(){ return message; }
4      }

```

25.7 *ClassC.java*

```

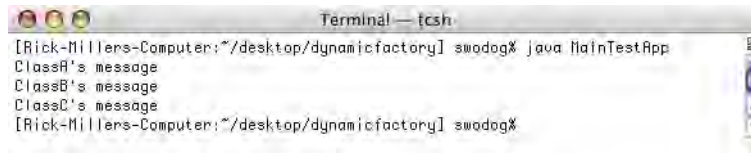
1      public class ClassC implements InterfaceType {
2          private String message = "ClassC's message";
3          public String getMessage(){ return message; }
4      }

```

```

1     public class MainTestApp {
2         public static void main(String[] args){
3             InterfaceType t1 = InterfaceTypeFactory.newObjectByClassName("ClassA");
4             InterfaceType t2 = InterfaceTypeFactory.newObjectByClassName("ClassB");
5             InterfaceType t3 = InterfaceTypeFactory.newObjectByClassName("ClassC");
6
7             System.out.println(t1.getMessage());
8             System.out.println(t2.getMessage());
9             System.out.println(t3.getMessage());
10        }
11    }

```



```

Terminal — tcsh
[rick-millers-computer:~/desktop/dynamicfactory] swodog$ java MainTestApp
ClassA's message
ClassB's message
ClassC's message
[rick-millers-computer:~/desktop/dynamicfactory] swodog$

```

Figure 25-1: Results of Running Example 25.8

ADVANTAGES OF THE DYNAMIC FACTORY PATTERN

One of the primary advantages of the Dynamic Factory pattern is that certain enhancements to an application that uses a dynamic factory can be made and implemented without the need to shut down the application. This can be done by making the necessary changes to any of the classes that are dynamically loaded and dropping them into the classpath as an upgrade to the previous version of that class. The next time the class is dynamically loaded the change will be effective. You will also have to structure your application architecture in such a way as to limit the number of outstanding references to the old version of the class otherwise the change will not propagate completely through the application.

To achieve this type of dynamic class loading behavior, the Dynamic Factory pattern must be used in conjunction with a properties file. Although it appears from looking at example 25.3 that the classes are being loaded each time the `newObjectByClassName()` method is called they are not. This is because the system class loader will not reload a class having the same name once it has been loaded into the JVM.

Quick Review

The Factory pattern is used to create classes whose purpose is to create objects of a specified type. The Dynamic Factory can be used to create objects via Java's dynamic class loading mechanism. One of the primary advantages of the Dynamic Factory pattern is that certain enhancements to an application that uses a dynamic factory can be made and implemented without the need to shut down the application.

THE MODEL-VIEW-CONTROLLER PATTERN

The Model-View-Controller (MVC) pattern is used to separate the visual representation of an application object from the application object itself. The MVC pattern consists of three primary components: 1) the *model*, which can consist of one or more classes working together to realize the functionality of a particular application, 2) the *view*, which can consist of one or more classes working together to implement the visual representation of the model, (*For example, the view could provide a user interface on the model's behalf.*), and 3) the *controller*, which can consist of one or more classes working together to coordinate messaging between the model and the view.

The approach I like to take when implementing the MVC pattern is to completely isolate the model from the view. In other words, the model should know nothing about the view, and the view should know nothing about the model. The controller functions as a liaison between the model and view components, coordinating intercomponent messaging between the two. This state of affairs is illustrated in figure 25-2.

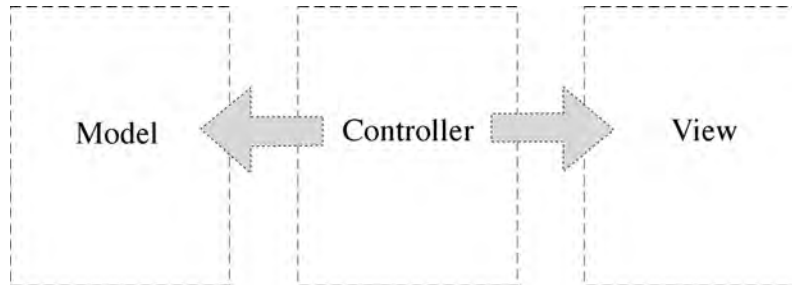


Figure 25-2: Model-View-Controller Pattern

The model is the most independent component in the MVC relationship. It should provide an interface and make no assumptions about the existence of the view or the controller. The view, especially if it's a GUI, may or may not need to know something about the controller. Assume for a moment that the controller implements the `ActionListener` interface with the intention of handling view component actions with the `actionPerformed()` method. The view would need a reference to an `ActionListener` but not to a controller per se.

Examples 25.9 through 25.11 gives the code for a simple implementation of the MVC pattern. I call this application Motivational Messages. When the application runs it presents a simple user interface consisting of a `Canvas` and a `JButton`. These components are contained in a class named `View`. When the button is clicked the `Controller` object handles the click and calls a method on the `Model` object to get the next message and on the `View` object to set the message. This is an example of interobject message coordination provided by a controller object. The `Controller` class in this case also serves as the application. The results of running this program are shown in figure 25-3.

```

1      public class Model {
2
3          private int i = 0;
4          private String[] messages = { "Eat right, get plenty of rest, and exercise daily.",
5                                       "Make love not war.",
6                                       "Carpe Diem!",
7                                       "Eat your vegetables.",
8                                       "Brush and floss your teeth three times daily.",
9                                       "A penny saved is a penny earned.",
10                                      "What you do today prepares you for tomorrow.",
11                                      "All work and no play makes Jack a dull boy." };
12
13         public String getMessage(){
14             if(i++ == (messages.length-1)) i = 0;
15             return messages[i];
16         }
17     } // end model class
  
```

25.9 Model.java

```

1      import javax.swing.*;
2      import java.awt.event.*;
3      import java.awt.*;
4
5      public class View extends JFrame {
6
7          private Canvas canvas = null;
8          private JButton button = null;
9          private String message = "";
10
11         public View(ActionListener al){
12             super("Motivational Messages");
13             canvas = new Canvas(){
14                 public void paint(Graphics g){
15                     g.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
16                     g.setFont(new Font("Roman", Font.BOLD, 14));
17                     g.drawString(message, 10, 20);
18                 } // end paint() method
19             }; // end Canvas inner class definition
20             button = new JButton("Next Message");
21             button.addActionListener(al);
22             this.getContentPane().setLayout(new BorderLayout());
23             this.getContentPane().add(canvas);
24             this.getContentPane().add(BorderLayout.SOUTH, button);
25             this.setSize(400, 100);
26             this.show();
  
```

25.10 View.java

```

27     }
28
29     public void setMessage(String message){
30         this.message = message;
31         canvas.repaint();
32     }
33
34 } // end View clas definition

```

```

1  import java.awt.event.*;
2
3  public class Controller implements ActionListener {
4
5      private Model its_model = null;
6      private View its_view = null;
7
8      public Controller(){
9          its_model = new Model();
10         its_view = new View(this);
11     }
12
13     public void actionPerformed(ActionEvent ae){
14         if(ae.getActionCommand().equals("Next Message")){
15             its_view.setMessage(its_model.getMessage());
16         }
17     }
18
19     public static void main(String[] args){
20         new Controller();
21     }
22 } // end Controller class definition

```

25.11 Controller.java



Figure 25-3: Results of Running Example 25.11 and Clicking the “Next Message” Button Several Times

POTENTIAL ISSUES WITH ONE ACTIONLISTENER

The MVC pattern does a nice job of helping you separate the concerns of an application from the concerns of its user interface. However, having the controller serve as the sole `ActionListener` for a large application will cause its `ActionPerformed()` method to grow quite large and ungainly due to the many `if-else` statements required to determine which GUI component was clicked. This issue can be effectively addressed with the help of the Command pattern which is discussed in the next section.

Quick Review

The Model-View-Controller (MVC) pattern is used to separate the visual representation of an application object from the application object itself. The MVC pattern consists of three primary components: 1) the *model*, which can consist of one or more classes working together to realize the functionality of a particular application, 2) the *view*, which can consist of one or more classes working together to implement the visual representation of the model, and 3) the *controller*, which can consist of one or more classes working together to coordinate messaging between the model and the view.

THE COMMAND PATTERN

The Command pattern is used to 1) decouple the knowledge of a particular action from an object that needs the action carried out, and 2) to encapsulate the action in the form of an object.

The Command pattern manifests itself to a certain degree in the Component/ActionListener relationship as you saw in the previous section. Components such as JButtons can add any number of ActionListeners. When a button is clicked an ActionEvent object is created and passed as an argument to an ActionListener's actionPerformed() method. If only one ActionListener exists to handle all button clicks then it's the responsibility of the actionPerformed() method to determine the source component and perform the necessary actions. As was noted above, in large application this can lead to a large actionPerformed() method.

Another approach to implementing the Command pattern in Java that you see frequent examples of is to extend components like JButton and JMenuItem and make them commands. I don't like this approach because it violates Coad's criteria. (*i.e.*, *Is a Command really a JButton?*)

The Command pattern implementation strategy I prefer is to combine the Dynamic Factory pattern with a separate Command class hierarchy. At the root of the hierarchy is an abstract class I will call BaseCommand and is given in example 25.12.

25.12 BaseCommand.java

```

1      package com.pulpfreepress.commands;
2
3      import com.pulpfreepress.interfaces.*;
4
5      public abstract class BaseCommand {
6
7          protected static iModel its_model = null;
8          protected static iView its_view = null;
9
10         public void setModel(iModel model) {
11             if(its_model == null){
12                 its_model = model;
13             }
14         }
15
16         public void setView(iView view) {
17             if(its_view == null){
18                 its_view = view;
19             }
20         }
21
22         public abstract void execute(); // must be implemented in derived classes
23
24     } // end BaseCommand class definition

```

Referring to example 25-12 — The BaseCommand class contains two protected static fields of type iModel and iView. These fields are protected so that subclasses can access them directly. Two methods setModel() and setView() initialize the its_model and its_view references accordingly. There is one abstract method named execute(). This method must be implemented by derived classes. It is the concrete command classes that implement the actions unique to each command. Some commands may interact with the model only while others may interact only with the view. Other commands may execute actions having nothing to do with the model or the view.

This approach to the Command pattern preserves the relationship between the MVC components as presented earlier but greatly simplifies the Controller's actionPerformed() method as is shown in example 25.13.

25.13 Controller.java

```

1      package com.pulpfreepress.controller;
2
3      import com.pulpfreepress.utils.*;
4      import com.pulpfreepress.commands.*;
5      import com.pulpfreepress.exceptions.*;
6      import com.pulpfreepress.model.*;
7      import com.pulpfreepress.view.*;
8      import com.pulpfreepress.interfaces.*;
9      import java.awt.event.*;
10
11     public class Controller implements ActionListener {
12
13         private CommandFactory command_factory = null;
14         private iModel its_model;
15         private iView its_view;
16
17         public Controller(){
18             command_factory = CommandFactory.getInstance();
19             its_model = new Model();
20             its_view = new View(this);
21         }

```

```

22
23
24     public void actionPerformed(ActionEvent ae){
25         try{
26             BaseCommand command = command_factory.getCommand(ae.getActionCommand());
27             command.setModel(its_model);
28             command.setView(its_view);
29             command.execute();
30         }catch(CommandNotFoundException cnfe){
31             System.out.println("Command not found!");
32         }
33     }
34
35     public static void main(String[] args){
36         new Controller();
37     } // end main() method
38 } // end Controller class definition

```

Referring to example 25.13 — the Controller class presented here is used in the comprehensive example presented in the next section so bear with me. Let’s focus on the actionPerformed() method beginning on line 24. The command_factory reference is used to dynamically load and create an instance of a command based on the name of the clicked component’s action command. (*A component’s action command is, by default, the string associated with the component’s label. For example, given a JButton with a label “Submit” its action command string is by default “Submit”.*) Once the command is created the setModel(), setView(), and execute() methods are called. So long as the command class exists and is in the system classpath things will work as expected.

Example 25.14 gives the code for the CommandFactory class.

25.14 CommandFactory.java

```

1     package com.pulpfreepress.utils;
2
3     import com.pulpfreepress.commands.*;
4     import com.pulpfreepress.exceptions.*;
5
6     public class CommandFactory {
7
8     private static CommandFactory command_factory_instance = null;
9     private static CommandProperties command_properties = null;
10
11     static {
12         command_properties = CommandProperties.getInstance();
13     }
14
15
16     private CommandFactory(){ }
17
18     public static CommandFactory getInstance(){
19         if(command_factory_instance == null){
20             command_factory_instance = new CommandFactory();
21         }
22         return command_factory_instance;
23     }
24
25     public BaseCommand getCommand(String command_string) throws CommandNotFoundException {
26         BaseCommand command = null;
27         if(command_string == null){
28             throw new CommandNotFoundException( command_string + " command class not found!");
29         } else{ try{
30             String command_classname = command_properties.getProperty(command_string);
31             Class command_class = Class.forName(command_classname);
32             Object command_object = command_class.newInstance();
33             command = (BaseCommand) command_object;
34         }catch(Throwable t){
35             t.printStackTrace();
36             throw new CommandNotFoundException(t.toString(), t);
37         }
38         } // end else
39         return command;
40     } // end getCommand() method
41
42 } // end CommandFactory class definition

```

Referring to example 25.14 — The CommandFactory implements the Singleton pattern. It also utilizes the services of the CommandProperties class which was presented earlier in example 25.1. The component action command strings are mapped to their respective command classes in the *Command.properties* file which I will show you shortly. Most of the work of this class is done in the getCommand() method. Here the command_string argument is

used to look up the class name of the command class that will actually perform the work. If the class exists it is loaded. If it loads successfully an instance of the class is created and returned.

Example 25.15 shows the contents of the `command.properties` file.

25.15 Command.properties file contents

```

1      #CommandProperties File - Edit Carefully
2      #Sat Jun 18 12:24:38 EDT 2005
3      PROPERTIES_FILE=Command.properties
4      Load=com.pulpfreepress.commands.LoadEmployeesCommand
5      Exit=com.pulpfreepress.commands.ApplicationExitCommand
6      NewSalariedEmployee=com.pulpfreepress.commands.NewSalariedEmployeeCommand
7      EditEmployee=com.pulpfreepress.commands.EditEmployeeCommand
8      Sort=com.pulpfreepress.commands.SortEmployeesCommand
9      DeleteEmployee=com.pulpfreepress.commands.DeleteEmployeeCommand
10     List=com.pulpfreepress.commands.ListEmployeesCommand
11     NewHourlyEmployee=com.pulpfreepress.commands.NewHourlyEmployeeCommand
12     Save=com.pulpfreepress.commands.SaveEmployeesCommand

```

Referring to example 25.15 — notice how command names like `Load` are mapped to their respective class handler.

Quick Review

The Command pattern is used to 1) decouple the knowledge of a particular action from an object that needs the action carried out, and 2) to encapsulate the action in the form of an object. The Command pattern can be combined with the Dynamic Factory pattern to map command names to class handlers and dynamically load and execute the command handler.

A COMPREHENSIVE PATTERN-BASED EXAMPLE

This section presents the design and code for a comprehensive example application that utilizes all the patterns discussed in this chapter. The application presented here allows you to create, edit, and delete hourly and salaried employees. It also lets you sort employees and save and retrieve employee data to and from disk. Figure 25-4 shows the UML class diagram for this application.

CODE LISTING BY PACKAGE NAME

This section gives the complete code listing for the application classes organized by package name.

com.pulpfreepress.commands

25.16 BaseCommand.java

```

1      package com.pulpfreepress.commands;
2
3      import com.pulpfreepress.interfaces.*;
4
5      public abstract class BaseCommand {
6          protected static iModel its_model = null;
7          protected static iView its_view = null;
8
9          public void setModel(iModel model){
10             if(its_model == null){
11                 its_model = model;
12             }
13         }
14
15         public void setView(iView view){
16             if(its_view == null){
17                 its_view = view;
18             }
19         }
20
21         public abstract void execute(); // must be implemented in derived classes
22     }

```

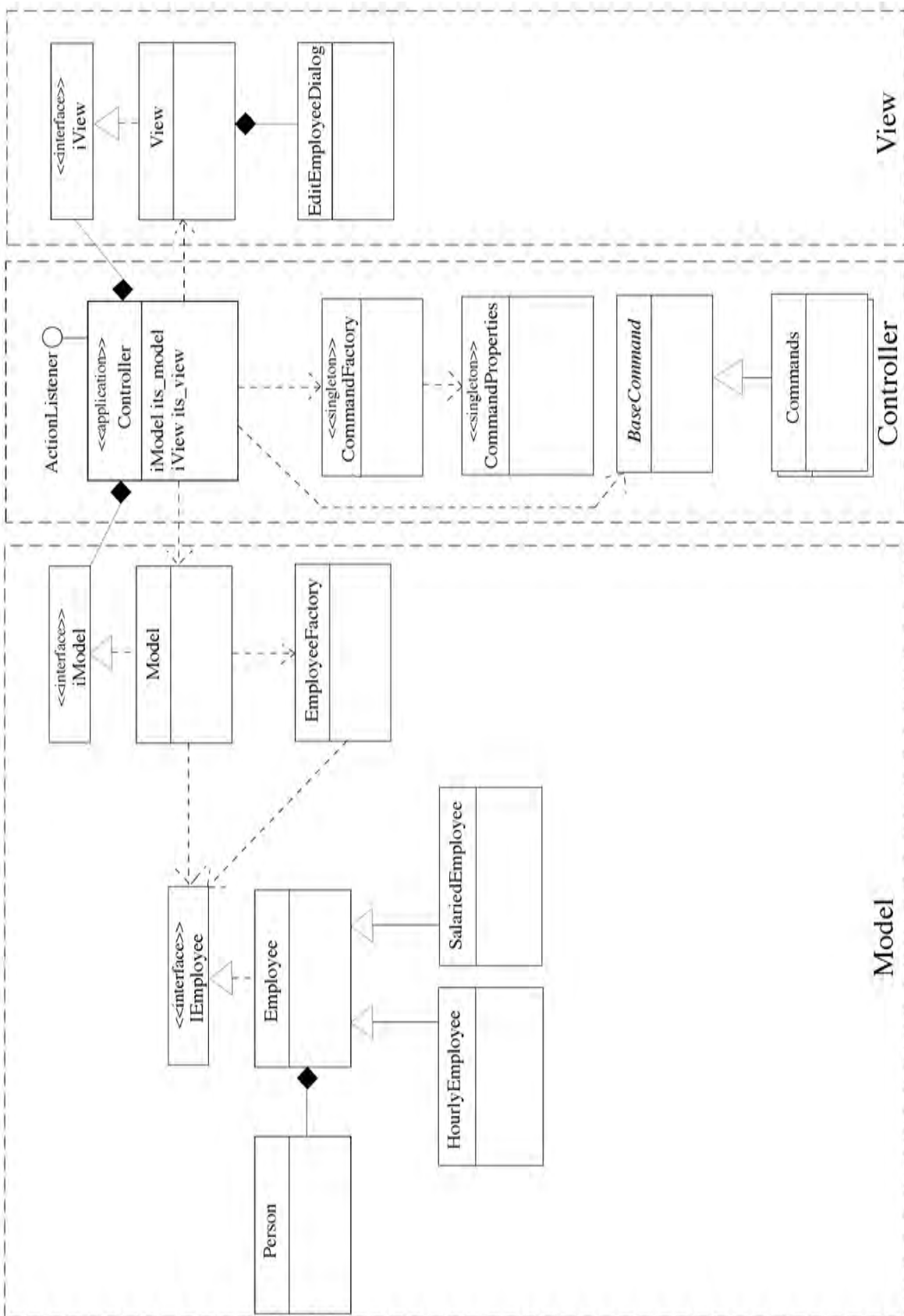


Figure 25-4: Employee Management Application UML Class Diagram

```
23     } // end BaseCommand class definition
```

25.17 ApplicationExitCommand.java

```
1     package com.pulpfreepress.commands;
2
3     public class ApplicationExitCommand extends BaseCommand {
4         public void execute(){
5             System.exit(0);
6         }
7     }
```

25.18 DeleteEmployeeCommand.java

```
1     package com.pulpfreepress.commands;
2
3     public class DeleteEmployeeCommand extends BaseCommand {
4
5         public void execute(){
6             if((its_model != null) && (its_view != null)){
7                 String employee_number = its_view.getDeleteEmployeeInfo();
8                 if(employee_number != null) {
9                     its_model.deleteEmployee(its_view.getDeleteEmployeeInfo());
10                }
11            }
12            its_view.displayEmployeeInfo(its_model.getAllEmployeesInfo());
13        } // end execute() method
14    } // end DeleteEmployeeCommand class definition
```

25.19 EditEmployeeCommand.java

```
1     package com.pulpfreepress.commands;
2
3     public class EditEmployeeCommand extends BaseCommand {
4
5         public void execute(){
6             if((its_model != null) && (its_view != null)){
7                 String[] emp_info = its_view.getEditEmployeeInfo();
8                 try{
9                     its_model.editEmployeeInfo(emp_info[0], emp_info[1], emp_info[2],
10                                                Integer.parseInt(emp_info[4]), Integer.parseInt(emp_info[5]),
11                                                Integer.parseInt(emp_info[6]), emp_info[3], emp_info[7]);
12                }catch(Throwable t){
13                    System.out.println("Invalid employee information - employee edit canceled!");
14                }
15            }
16            its_view.displayEmployeeInfo(its_model.getAllEmployeesInfo());
17        } // end execute() method
18    } // end EditEmployeeCommand class definition
```

25.20 ListEmployeesCommand.java

```
1     package com.pulpfreepress.commands;
2
3     public class ListEmployeesCommand extends BaseCommand {
4         public void execute(){
5             if(its_model != null){
6                 its_view.displayEmployeeInfo(its_model.getAllEmployeesInfo());
7             }
8         }
9     } // end ListEmployeesCommand class definition
```

25.21 LoadEmployeesCommand.java

```
1     package com.pulpfreepress.commands;
2
3     public class LoadEmployeesCommand extends BaseCommand {
4
5         public void execute(){
6             if((its_model != null) && (its_view != null)){
7                 its_model.loadEmployeesFromFile(its_view.getLoadFile());
8             }
9         }
10        its_view.displayEmployeeInfo(its_model.getAllEmployeesInfo());
11    } // end execute() method
12 } // end DeleteEmployeeCommand class definition
```

25.22 NewHourlyEmployeeCommand.java

```
1     package com.pulpfreepress.commands;
2
3     public class NewHourlyEmployeeCommand extends BaseCommand {
```

```

4     public void execute(){
5         if(its_model != null){
6             String[] emp_info = its_view.getNewHourlyEmployeeInfo();
7             try{
8                 its_model.createHourlyEmployee(emp_info[0], emp_info[1], emp_info[2],
9                                                 Integer.parseInt(emp_info[4]), Integer.parseInt(emp_info[5]),
10                                                Integer.parseInt(emp_info[6]), emp_info[3], emp_info[7],
11                                                Double.parseDouble(emp_info[8]), Double.parseDouble(emp_info[9]));
12             }catch(Throwable t){
13                 System.out.println("Invalid employee information - employee not created!");
14             }
15             its_view.displayEmployeeInfo(its_model.getAllEmployeesInfo());
16         }
17     } // end execute() method
18 } // end NewHourlyEmployeeCommand class definition

```

25.23 NewSalariedEmployeeCommand.java

```

1     package com.pulpfreepress.commands;
2
3     public class NewSalariedEmployeeCommand extends BaseCommand {
4         public void execute(){
5             if(its_model != null){
6                 String[] emp_info = its_view.getNewSalariedEmployeeInfo();
7                 try{
8                     its_model.createSalariedEmployee(emp_info[0], emp_info[1], emp_info[2],
9                                                       Integer.parseInt(emp_info[4]), Integer.parseInt(emp_info[5]),
10                                                      Integer.parseInt(emp_info[6]), emp_info[3], emp_info[7],
11                                                      Double.parseDouble(emp_info[10]));
12                 }catch(Throwable t){
13                     System.out.println("Invalid employee information - employee not created!");
14                 }
15                 its_view.displayEmployeeInfo(its_model.getAllEmployeesInfo());
16             }
17         } // end execute() method
18     } // end NewSalariedEmployeeCommand class definition

```

25.24 SaveEmployeesCommand.java

```

1     package com.pulpfreepress.commands;
2
3     public class SaveEmployeesCommand extends BaseCommand {
4
5         public void execute(){
6             if((its_model != null) && (its_view != null)){
7                 its_model.saveEmployeesToFile(its_view.getSaveFile());
8             }
9         }
10        its_view.displayEmployeeInfo(its_model.getAllEmployeesInfo());
11    } // end execute() method
12 } // end DeleteEmployeeCommand class definition

```

25.25 SortEmployeesCommand.java

```

1     package com.pulpfreepress.commands;
2
3     public class SortEmployeesCommand extends BaseCommand {
4         public void execute(){
5             if(its_model != null){
6                 its_model.sortEmployees();
7                 its_view.displayEmployeeInfo(its_model.getAllEmployeesInfo());
8             }
9         } // end execute() method
10    } // end SortEmployeesCommand class definition

```

com.pulpfreepress.controller

25.26 Controller.java

```

1     package com.pulpfreepress.controller;
2
3     import com.pulpfreepress.utils.*;
4     import com.pulpfreepress.commands.*;
5     import com.pulpfreepress.exceptions.*;
6     import com.pulpfreepress.model.*;
7     import com.pulpfreepress.view.*;
8     import com.pulpfreepress.interfaces.*;
9     import java.awt.event.*;
10

```

```

11     public class Controller implements ActionListener {
12
13         private CommandFactory command_factory = null;
14         private iModel its_model;
15         private iView its_view;
16
17         public Controller(){
18             command_factory = CommandFactory.getInstance();
19             its_model = new Model();
20             its_view = new View(this);
21         }
22
23
24         public void actionPerformed(ActionEvent ae){
25             try{
26                 BaseCommand command = command_factory.getCommand(ae.getActionCommand());
27                 command.setModel(itself_model);
28                 command.setView(itself_view);
29                 command.execute();
30             }catch(CommandNotFoundException cnfe){
31                 System.out.println("Command not found!");
32             }
33         }
34
35         public static void main(String[] args){
36             new Controller();
37         } // end main() method
38     } // end Controller class definition

```

COM.PULPFREEPRESS.EXCEPTIONS

25.27 *CommandNotFoundException.java*

```

1     package com.pulpfreepress.exceptions;
2
3     public class CommandNotFoundException extends Exception {
4
5         public CommandNotFoundException(String message, Throwable ex){
6             super(message, ex);
7         }
8
9         public CommandNotFoundException(String message){
10            super(message);
11        }
12
13
14        public CommandNotFoundException(){
15            this("Command Not Found Exception");
16        }
17    } // end CommandNotFoundException class definition

```

COM.PULPFREEPRESS.INTERFACES

25.28 *iModel.java*

```

1     package com.pulpfreepress.interfaces;
2
3     import java.util.*;
4     import java.io.*;
5
6     public interface iModel {
7         public void createHourlyEmployee(String fname, String mname, String lname,
8             int dob_year, int dob_month, int dob_day,
9             String gender, String employee_number,
10            double hours_worked, double hourly_rate);
11
12        public void createSalariedEmployee(String fname, String mname, String lname,
13            int dob_year, int dob_month, int dob_day,
14            String gender, String employee_number,
15            double salary);
16
17        public void editEmployeeInfo(String fname, String mname, String lname,
18            int dob_year, int dob_month, int dob_day,
19            String gender, String employee_number);
20
21    }

```

```

22     public String[] getAllEmployeesInfo();
23     public String getEmployeeInfo(String employee_number);
24     public void sortEmployees();
25     public void deleteEmployee(String employee_number);
26     public void saveEmployeesToFile(File file);
27     public void loadEmployeesFromFile(File file);
28 } // end iModel interface definition

```

25.29 *iView.java*

```

1     package com.pulpfreepress.interfaces;
2     import java.io.*;
3
4     public interface iView {
5         public void displayEmployeeInfo(String[] employees_info);
6         public String[] getNewHourlyEmployeeInfo();
7         public String[] getNewSalariedEmployeeInfo();
8         public String[] getEditEmployeeInfo();
9         public String getDeleteEmployeeInfo();
10        public File getSaveFile();
11        public File getLoadFile();
12
13    } // end iView interface definition

```

COM.PULPFREEPRESS.MODEL25.30 *IEmployee.java*

```

1     package com.pulpfreepress.model;
2
3     import java.util.*;
4
5     public interface IEmployee {
6         int getAge();
7         String getFullName();
8         String getNameAndAge();
9         String getFirstName();
10        String getMiddleName();
11        String getLastName();
12        String getGender();
13        String getEmployeeNumber();
14        void setBirthday(int year, int month, int day);
15        Calendar getBirthday();
16        void setFirstName(String f_name);
17        void setMiddleName(String m_name);
18        void setLastName(String l_name);
19        void setGender(String gender);
20        void setEmployeeNumber(String emp_no);
21        void setPayInfo(PayInfo pi);
22        double getPay();
23        String toString();
24    }

```

25.31 *Employee.java*

```

1     package com.pulpfreepress.model;
2
3     import java.util.*;
4     import java.io.*;
5
6     public abstract class Employee implements IEmployee, Cloneable, Comparable, Serializable {
7         private Person _person = null;
8         private String _employee_number = null;
9
10        protected Employee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
11            int dob_day, String gender, String employee_number){
12            _person = new Person(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender);
13            _employee_number = employee_number;
14        } // end constructor
15
16        public int getAge() { return _person.getAge(); }
17        public String getFullName() { return _person.getFullName(); }
18        public String getNameAndAge() { return _person.getNameAndAge(); }
19        public String getFirstName() { return _person.getFirstName(); }
20        public String getMiddleName() { return _person.getMiddleName(); }
21        public String getLastName() { return _person.getLastName(); }
22        public String getGender() { return _person.getGender(); }
23        public String getEmployeeNumber() { return _employee_number; }
24        public void setBirthday(int year, int month, int day) { _person.setBirthday(year, month, day); }
25        public Calendar getBirthday() { return _person.getBirthday(); }

```



```

26     public void setFirstName(String f_name) { _person.setFirstName(f_name); }
27     public void setMiddleName(String m_name) { _person.setMiddleName(m_name); }
28     public void setLastName(String l_name) { _person.setLastName(l_name); }
29     public void setGender(String gender) { _person.setGender(gender); }
30     public void setEmployeeNumber(String emp_no) { _employee_number = emp_no; }
31
32     public String toString(){ return _person.toString() + " " + _employee_number; }
33
34     public boolean equals(Object o){
35         if(o == null) return false;
36         boolean is_equal = false;
37         if(o instanceof Employee){
38             if(this.toString().equals(o.toString())){
39                 is_equal = true;
40             }
41         }
42         return is_equal;
43     }
44
45     public int hashCode(){
46         return this.toString().hashCode();
47     }
48
49
50     public int compareTo(Object o){
51         return this.toString().compareTo(o.toString());
52     }
53
54     public abstract void setPayInfo(PayInfo pi); // defer implementation
55     public abstract double getPay(); // defer implementation
56
57 } // end Employee class definition

```

25.32 Person.java

```

1     package com.pulpfreepress.model;
2
3     import java.util.*;
4     import java.io.*;
5
6     public class Person implements Cloneable, Comparable, Serializable {
7         private String first_name = null;
8         private String middle_name = null;
9         private String last_name = null;
10        private Calendar birthday = null;
11        private String gender = null;
12
13        public static final String MALE = "Male";
14        public static final String FEMALE = "Female";
15
16        public Person(String f_name, String m_name, String l_name, int dob_year, int dob_month,
17                    int dob_day, String gender){
18            first_name = f_name;
19            middle_name = m_name;
20            last_name = l_name;
21            this.gender = gender;
22
23            birthday = Calendar.getInstance();
24            birthday.set(dob_year, dob_month, dob_day);
25        }
26
27        public int getAge(){
28            Calendar today = Calendar.getInstance();
29            int now = today.get(Calendar.YEAR);
30            int then = birthday.get(Calendar.YEAR);
31            return (now - then);
32        }
33
34        public String getFullName(){ return (first_name + " " + middle_name + " " + last_name); }
35
36        public String getFirstName(){ return first_name; }
37        public void setFirstName(String f_name) { first_name = f_name; }
38
39        public String getMiddleName(){ return middle_name; }
40        public void setMiddleName(String m_name){ middle_name = m_name; }
41
42        public String getLastName(){ return last_name; }
43        public void setLastName(String l_name){ last_name = l_name; }
44
45        public String getNameAndAge(){ return (getFullName() + " " + getAge()); }
46

```

```

47     public String getGender(){ return gender; }
48     public void setGender(String gender){ this.gender = gender; }
49
50     public void setBirthday(int year, int month, int day){ birthday.set(year, month, day); }
51     public Calendar getBirthday(){ return birthday; }
52
53     public String toString(){
54         return this.getFullName() + " " + gender + " " + birthday.get(Calendar.DATE) + "/"
55             + birthday.get(Calendar.MONTH) + "/" + birthday.get(Calendar.YEAR);
56     }
57
58     public boolean equals(Object o){
59         if(o == null) return false;
60         boolean is_equal = false;
61         if(o instanceof Person){
62             if(this.first_name.equals(((Person)o).first_name) &&
63                 this.middle_name.equals(((Person)o).middle_name) &&
64                 this.last_name.equals(((Person)o).last_name) && this.gender.equals(((Person)o).gender) &&
65                 (this.birthday.get(Calendar.YEAR) == ((Person)o).birthday.get(Calendar.YEAR)) &&
66                 (this.birthday.get(Calendar.MONTH) == ((Person)o).birthday.get(Calendar.MONTH)) &&
67                 (this.birthday.get(Calendar.DATE) == ((Person)o).birthday.get(Calendar.DATE)) ){
68                 is_equal = true;
69             }
70         }
71         return is_equal;
72     }
73
74     public int hashCode(){
75         return this.toString().hashCode();
76     }
77
78     public Object clone() throws CloneNotSupportedException {
79         super.clone();
80         return new Person(new String(first_name), new String(middle_name), new String(last_name),
81             birthday.get(Calendar.YEAR), birthday.get(Calendar.MONTH),
82             birthday.get(Calendar.DATE), new String(gender));
83     }
84
85     public int compareTo(Object o){
86         return this.toString().compareTo(o.toString());
87     }
88
89 } //end Person class

```

25.33 *HourlyEmployee.java*

```

1     package com.pulpfreepress.model;
2
3     import java.util.*;
4     import java.io.*;
5     import java.text.*;
6
7     public class HourlyEmployee extends Employee implements Cloneable, Comparable, Serializable{
8         private double _hours_worked = 0;
9         private double _hourly_rate = 0.0;
10
11         public HourlyEmployee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
12             int dob_day, String gender, String employee_number){
13             super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender, employee_number);
14         }
15
16         public void setPayInfo(PayInfo pi){
17             _hours_worked = pi.getHoursWorked();
18             _hourly_rate = pi.getHourlyRate();
19         }
20
21         public double getPay() { return _hours_worked * _hourly_rate; }
22
23         public String toString() {
24             NumberFormat pay_format = NumberFormat.getInstance();
25             return super.toString() + " $" + pay_format.format(getPay());
26         }
27     }
28
29     public Object clone() throws CloneNotSupportedException {
30         super.clone();
31         return new HourlyEmployee(new String(getFirstName()), new String(getMiddleName()),
32             new String(getLastName()), getBirthday().get(Calendar.YEAR),
33             getBirthday().get(Calendar.MONTH), getBirthday().get(Calendar.DATE),
34             new String(getGender()), new String(getEmployeeNumber()));
35     }

```

```

36
37     public boolean equals(Object o){
38         if(o == null) return false;
39         boolean is_equal = false;
40         if(o instanceof HourlyEmployee){
41             if(this.toString().equals(o.toString())){
42                 is_equal = true;
43             }
44         }
45         return is_equal;
46     }
47
48     public int hashCode(){
49         return this.toString().hashCode();
50     }
51
52
53     public int compareTo(Object o){
54         return this.toString().compareTo(o.toString());
55     }
56
57
58 } // end HourlyEmployee class definition

```

25.34 SalariedEmployee.java

```

1     package com.pulpfreepress.model;
2
3     import java.util.*;
4     import java.io.*;
5     import java.text.*;
6
7     public class SalariedEmployee extends Employee implements Cloneable, Comparable, Serializable {
8         private double _salary = 0;
9
10        public SalariedEmployee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
11                               int dob_day, String gender, String employee_number){
12            super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender, employee_number);
13        }
14
15        public void setPayInfo(PayInfo pi){
16            _salary = pi.getSalary();
17        }
18
19        public double getPay() { return (_salary/12.0)/2.0; }
20
21        public String toString() {
22            NumberFormat pay_format = NumberFormat.getInstance();
23            return super.toString() + " $" + pay_format.format(getPay());
24        }
25
26
27        public Object clone() throws CloneNotSupportedException {
28            super.clone();
29            return new SalariedEmployee(new String(getFirstName()), new String(getMiddleName()),
30                                       new String(getLastName()), getBirthday().get(Calendar.YEAR),
31                                       getBirthday().get(Calendar.MONTH), getBirthday().get(Calendar.DATE),
32                                       new String(getGender()), new String(getEmployeeNumber()));
33        }
34
35        public boolean equals(Object o){
36            if(o == null) return false;
37            boolean is_equal = false;
38            if(o instanceof SalariedEmployee){
39                if(this.toString().equals(o.toString())){
40                    is_equal = true;
41                }
42            }
43            return is_equal;
44        }
45
46        public int hashCode(){
47            return this.toString().hashCode();
48        }
49
50
51        public int compareTo(Object o){
52            return this.toString().compareTo(o.toString());
53        }
54    } // end SalariedEmployee class definition

```

25.35 PayInfo.java

```

1     package com.pulpfreepress.model;
2
3     public class PayInfo {
4         private double _salary = 0;
5         private double _hours_worked = 0;
6         private double _hourly_rate = 0;
7
8         public PayInfo(double salary){
9             _salary = salary;
10        }
11
12        public PayInfo(double hours_worked, double hourly_rate){
13            _hours_worked = hours_worked;
14            _hourly_rate = hourly_rate;
15        }
16
17        public PayInfo(){ }
18
19        public double getHoursWorked(){ return _hours_worked; }
20        public double getHourlyRate(){ return _hourly_rate; }
21        public double getSalary(){ return _salary; }
22
23    } // end PayInfo class definition

```

25.36 IEmployeeFactory.java

```

1     package com.pulpfreepress.model;
2
3     public interface IEmployeeFactory {
4         IEmployee getNewSalariedEmployee(String f_name, String m_name, String l_name, int dob_year,
5             int dob_month, int dob_day, String gender, String employee_number);
6         IEmployee getNewHourlyEmployee(String f_name, String m_name, String l_name, int dob_year,
7             int dob_month, int dob_day, String gender, String employee_number);
8     }

```

25.37 EmployeeFactory.java

```

1     package com.pulpfreepress.model;
2
3     public class EmployeeFactory implements IEmployeeFactory {
4
5         public IEmployee getNewSalariedEmployee(String f_name, String m_name, String l_name,
6             int dob_year, int dob_month, int dob_day, String gender,
7             String employee_number){
8             return new SalariedEmployee(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender,
9                 employee_number);
10        }
11
12        public IEmployee getNewHourlyEmployee(String f_name, String m_name, String l_name, int dob_year,
13            int dob_month, int dob_day, String gender,
14            String employee_number){
15            return new HourlyEmployee(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender,
16                employee_number);
17        }
18    } // end EmployeeFactory class definition

```

25.38 Model.java

```

1     package com.pulpfreepress.model;
2
3     import com.pulpfreepress.interfaces.*;
4     import java.util.*;
5     import java.io.*;
6
7     public class Model implements iModel {
8
9         private List employee_list = null;
10        private IEmployeeFactory employee_factory = null;
11
12        public Model(){
13            employee_list = new LinkedList();
14            employee_factory = new EmployeeFactory();
15        }
16
17        public void createHourlyEmployee(String fname, String mname, String lname,
18            int dob_year, int dob_month, int dob_day,
19            String gender, String employee_number,
20            double hours_worked, double hourly_rate){
21
22            PayInfo pay_info = new PayInfo(hours_worked, hourly_rate);

```

```

23         IEmployee employee = employee_factory.getNewHourlyEmployee(fname, mname, lname, dob_year,
24                                                     dob_month, dob_day, gender,
25                                                     employee_number);
26         employee.setPayInfo(pay_info);
27         employee_list.add(employee);
28
29     }
30
31
32     public void createSalariedEmployee(String fname, String mname, String lname,
33                                       int dob_year, int dob_month, int dob_day,
34                                       String gender, String employee_number,
35                                       double salary){
36
37         PayInfo pay_info = new PayInfo(salary);
38         IEmployee employee = employee_factory.getNewSalariedEmployee(fname, mname, lname, dob_year,
39                                                     dob_month, dob_day, gender,
40                                                     employee_number);
41         employee.setPayInfo(pay_info);
42         employee_list.add(employee);
43
44     }
45
46
47     public void editEmployeeInfo(String fname, String mname, String lname,
48                                 int dob_year, int dob_month, int dob_day,
49                                 String gender, String employee_number){
50
51         IEmployee employee = null;
52         for(Iterator it = employee_list.iterator(); it.hasNext();){
53             employee = (IEmployee)it.next();
54             if(employee.getEmployeeNumber().equals(employee_number)) break;
55         }
56
57         employee.setFirstName(fname);
58         employee.setMiddleName(mname);
59         employee.setLastName(lname);
60         employee.setBirthDay(dob_year, dob_month, dob_day);
61     }
62
63
64
65     public String[] getAllEmployeesInfo(){
66         String[] emp_info = new String[employee_list.size()];
67         Iterator it = employee_list.iterator();
68         for(int i = 0; it.hasNext();){
69             emp_info[i++] = it.next().toString();
70         }
71         return emp_info;
72     }
73
74     public String getEmployeeInfo(String employee_number){
75         IEmployee employee = null;
76         for(Iterator it = employee_list.iterator(); it.hasNext();){
77             employee = (IEmployee)it.next();
78             if(employee.getEmployeeNumber().equals(employee_number)) break;
79         }
80         return employee.toString();
81     }
82
83     public void sortEmployees(){
84         Collections.sort(employee_list);
85     }
86
87     public void deleteEmployee(String employee_number){
88         IEmployee employee = null;
89         for(Iterator it = employee_list.iterator(); it.hasNext();){
90             employee = (IEmployee)it.next();
91             if(employee.getEmployeeNumber().equals(employee_number)){
92                 employee_list.remove(employee);
93                 break;
94             }
95         }
96     }
97
98     public void saveEmployeesToFile(File file){
99         if(file == null){
100             file = new File("employees.dat");
101         }
102
103         FileOutputStream fos = null;

```

```

104     ObjectOutputStream oos = null;
105
106     try{
107         fos = new FileOutputStream(file);
108         oos = new ObjectOutputStream(fos);
109         oos.writeObject(employee_list);
110         oos.close();
111     }catch(Exception e){
112         System.out.println("Problem saving employees file to disk!");
113     }
114 }
115
116 public void loadEmployeesFromFile(File file){
117     if(file == null){
118         file = new File("employees.dat");
119     }
120
121     FileInputStream fis = null;
122     ObjectInputStream ois = null;
123
124     try{
125         fis = new FileInputStream(file);
126         ois = new ObjectInputStream(fis);
127         employee_list = (LinkedList)ois.readObject();
128         ois.close();
129     }catch(Exception e){
130         System.out.println("Problem saving employees file to disk!");
131     }
132 }
133 }
134
135 } // end Model class definition

```

com.pulpfreepress.util

25.39 *CommandFactory.java*

```

1     package com.pulpfreepress.util;
2
3     import com.pulpfreepress.commands.*;
4     import com.pulpfreepress.exceptions.*;
5
6     public class CommandFactory {
7
8     private static CommandFactory command_factory_instance = null;
9     private static CommandProperties command_properties = null;
10
11     static {
12         command_properties = CommandProperties.getInstance();
13     }
14
15     private CommandFactory(){ }
16
17     public static CommandFactory getInstance(){
18         if(command_factory_instance == null){
19             command_factory_instance = new CommandFactory();
20         }
21         return command_factory_instance;
22     }
23
24     public BaseCommand getCommand(String command_string) throws CommandNotFoundException {
25         BaseCommand command = null;
26         if(command_string == null){
27             throw new CommandNotFoundException( command_string + " command class not found!");
28         } else{ try{
29             String command_classname = command_properties.getProperty(command_string);
30             Class command_class = Class.forName(command_classname);
31             Object command_object = command_class.newInstance();
32             command = (BaseCommand) command_object;
33         }catch(Throwable t){
34             t.printStackTrace();
35             throw new CommandNotFoundException(t.toString(), t);
36         }
37     } // end else
38     return command;
39 } // end getCommand() method
40 } // end CommandFactory class definition

```

```

1     package com.pulpfreepress.utils;
2
3     import java.util.*;
4     import java.io.*;
5
6     public class CommandProperties extends Properties {
7
8         // class constants - default key strings
9         public static final String PROPERTIES_FILE           = "PROPERTIES_FILE";
10        public static final String NEWHOURLYEMPLOYEE_COMMAND = "NewHourlyEmployee";
11        public static final String NEWSALARIEDEMPLOYEE_COMMAND = "NewSalariedEmployee";
12        public static final String EXIT_COMMAND              = "Exit";
13        public static final String LIST_COMMAND               = "List";
14        public static final String SORT_COMMAND              = "Sort";
15        public static final String SAVE_COMMAND              = "Save";
16        public static final String EDITEMPLOYEE_COMMAND      = "EditEmployee";
17        public static final String DELETEEMPLOYEE_COMMAND    = "DeleteEmployee";
18        public static final String LOAD_COMMAND              = "Load";
19
20        // class constants - default value strings
21        private static final String PROPERTIES_FILE_VALUE = "Command.properties";
22        private static final String NEWHOURLYEMPLOYEE_COMMAND_CLASSNAME=
23            "com.pulpfreepress.commands.NewHourlyEmployeeCommand";
24        private static final String NEWSALARIEDEMPLOYEE_COMMAND_CLASSNAME=
25            "com.pulpfreepress.commands.NewSalariedEmployeeCommand";
26        private static final String EXIT_COMMAND_CLASSNAME=
27            "com.pulpfreepress.commands.ApplicationExitCommand";
28        private static final String LIST_COMMAND_CLASSNAME=
29            "com.pulpfreepress.commands.ListEmployeesCommand";
30        private static final String SORT_COMMAND_CLASSNAME=
31            "com.pulpfreepress.commands.SortEmployeesCommand";
32        private static final String SAVE_COMMAND_CLASSNAME=
33            "com.pulpfreepress.commands.SaveEmployeesCommand";
34        private static final String EDITEMPLOYEE_COMMAND_CLASSNAME=
35            "com.pulpfreepress.commands.EditEmployeeCommand";
36        private static final String DELETEEMPLOYEE_COMMAND_CLASSNAME=
37            "com.pulpfreepress.commands.DeleteEmployeeCommand";
38        private static final String LOAD_COMMAND_CLASSNAME=
39            "com.pulpfreepress.commands.LoadEmployeesCommand";
40
41        // class variables
42        private static CommandProperties _properties_object = null;
43
44        private CommandProperties( String properties_file ){
45            try{
46                FileInputStream fis = new FileInputStream(properties_file);
47                load(fis);
48            }catch(Exception e) {
49                System.out.println("Problem opening properties file!");
50                System.out.println("Bootstrapping properties...");
51                try{
52                    FileOutputStream fos = new FileOutputStream(PROPERTIES_FILE_VALUE);
53                    setProperty(PROPERTIES_FILE, PROPERTIES_FILE_VALUE);
54                    setProperty(NEWHOURLYEMPLOYEE_COMMAND, NEWHOURLYEMPLOYEE_COMMAND_CLASSNAME);
55                    setProperty(NEWSALARIEDEMPLOYEE_COMMAND, NEWSALARIEDEMPLOYEE_COMMAND_CLASSNAME);
56                    setProperty(EXIT_COMMAND, EXIT_COMMAND_CLASSNAME);
57                    setProperty(LIST_COMMAND, LIST_COMMAND_CLASSNAME);
58                    setProperty(SORT_COMMAND, SORT_COMMAND_CLASSNAME);
59                    setProperty(SAVE_COMMAND, SAVE_COMMAND_CLASSNAME);
60                    setProperty(EDITEMPLOYEE_COMMAND, EDITEMPLOYEE_COMMAND_CLASSNAME);
61                    setProperty(DELETEEMPLOYEE_COMMAND, DELETEEMPLOYEE_COMMAND_CLASSNAME);
62                    setProperty(LOAD_COMMAND, LOAD_COMMAND_CLASSNAME);
63
64                    super.store(fos, "CommandProperties File - Edit Carefully");
65                    fos.close();
66                }catch(Exception e2){ System.out.println("Uh ohh...Bigger problems exist!"); }
67            }
68        }
69
70        /*****
71        * Private default constructor. Applications will get an instance via the getInstance() method.
72        * @see getInstance()
73        *****/
74        private CommandProperties(){
75            this(PROPERTIES_FILE_VALUE);
76        }
77
78        /*****
79        * The store() method attempts to persist its properties collection.

```

```

80      *****/
81      public void store(){
82          try{
83              FileOutputStream fos = new
84                  FileOutputStream(getProperty(PROPERTIES_FILE));
85              super.store(fos, "CommandProperties File");
86              fos.close();
87          }catch(Exception e){ System.out.println("Trouble storing properties!"); }
88      }
89
90      /*****
91      * getInstance() returns a singleton instance of the CommandProperties object.
92      *****/
93      public static CommandProperties getInstance(){
94          if(_properties_object == null){
95              _properties_object = new CommandProperties();
96          }
97          return _properties_object;
98      }
99  } // end CommandProperties class definition

```

com.pulpfreepress.view

25.41 View.java

```

1      package com.pulpfreepress.view;
2
3      import com.pulpfreepress.interfaces.*;
4      import java.awt.event.*;
5      import javax.swing.*;
6      import java.util.*;
7      import java.io.*;
8
9      public class View extends JFrame implements iView {
10
11          private JPanel panell = null;
12          private JTextArea textareal = null;
13          private int initial_frame_position_x = 50;
14          private int initial_frame_position_y = 50;
15          private int initial_frame_width = 400;
16          private int initial_frame_height = 400;
17          private int initial_textarea_rows = 20;
18          private int initial_textarea_columns = 20;
19          private EditEmployeeDialog edit_employee_dialog = null;
20          private String[] current_employee_list = null;
21          private JMenuItem delete_employee_menuitem = null;
22          private JMenuItem edit_employee_menuitem = null;
23
24          public View(ActionListener al){
25              this.setupGUI(al);
26          }
27
28          public void displayEmployeeInfo(String[] employees_info){
29              textareal.setText("");
30              for(int i = 0; i<employees_info.length; i++){
31                  textareal.append(employees_info[i] + "\n");
32              }
33              current_employee_list = employees_info;
34              if(current_employee_list.length == 0){
35                  this.delete_employee_menuitem.setEnabled(false);
36                  this.edit_employee_menuitem.setEnabled(false);
37              }else{
38                  this.delete_employee_menuitem.setEnabled(true);
39                  this.edit_employee_menuitem.setEnabled(true);
40              }
41          }
42
43          public String[] getNewHourlyEmployeeInfo(){
44              edit_employee_dialog.setTitle("New Hourly Employee");
45              edit_employee_dialog.clearTextFields();
46              edit_employee_dialog.showHourlyInfo();
47              edit_employee_dialog.hideSalaryInfo();
48              edit_employee_dialog.enableEmployeeNumberTextField();
49              edit_employee_dialog.setVisible(true);
50
51              return edit_employee_dialog.getEmployeeInfo();
52          }
53

```



```

54
55     public String[] getNewSalariedEmployeeInfo(){
56         edit_employee_dialog.setTitle("New Salaried Employee");
57         edit_employee_dialog.clearTextFields();
58         edit_employee_dialog.showSalaryInfo();
59         edit_employee_dialog.hideHourlyInfo();
60         edit_employee_dialog.enableEmployeeNumberTextField();
61         edit_employee_dialog.setVisible(true);
62         return edit_employee_dialog.getEmployeeInfo();
63     }
64
65     public String[] getEditEmployeeInfo(){
66         edit_employee_dialog.setTitle("Edit Employee Information");
67         edit_employee_dialog.clearTextFields();
68         edit_employee_dialog.hideHourlyAndSalaryInfo();
69         edit_employee_dialog.disableEmployeeNumberTextField();
70         String emp_info_string = current_employee_list[selectedLineNumber()];
71         String[] emp_string_array = employeeInfoStringToStringArray(emp_info_string);
72         edit_employee_dialog.populateTextFields(emp_string_array);
73         edit_employee_dialog.setVisible(true);
74         return edit_employee_dialog.getEditedEmployeeInfo();
75     }
76
77
78     public String getDeleteEmployeeInfo(){
79         String emp_number = "";
80         if(current_employee_list != null){
81             String emp_info_string = current_employee_list[selectedLineNumber()];
82             String[] emp_string_array = employeeInfoStringToStringArray(emp_info_string);
83             if(emp_string_array != null){
84                 emp_number = emp_string_array[7];
85             }
86         }
87         return emp_number;
88     }
89
90     private JMenuBar setupMenuBar(ActionListener al){
91         /** create menubar **/
92         JMenuBar menubar = new JMenuBar();
93
94         /** create file menu and menu items **/
95         JMenu file_menu = new JMenu("File");
96         JMenuItem file_loadEmployees_menuitem = new JMenuItem("Load Employees...");
97         file_loadEmployees_menuitem.setActionCommand("Load");
98         file_loadEmployees_menuitem.addActionListener(al);
99         JMenuItem file_saveEmployees_menuitem = new JMenuItem("Save Employees...");
100        file_saveEmployees_menuitem.setActionCommand("Save");
101        file_saveEmployees_menuitem.addActionListener(al);
102        JMenuItem file_exit_menuitem = new JMenuItem("Exit");
103        file_exit_menuitem.addActionListener(al);
104
105        file_menu.add(file_loadEmployees_menuitem);
106        file_menu.add(file_saveEmployees_menuitem);
107        file_menu.add(file_exit_menuitem);
108
109        menubar.add(file_menu);
110
111        /** create employees menu and menu items **/
112        JMenu employees_menu = new JMenu("Employees");
113        JMenuItem employees_list_menuitem = new JMenuItem("List");
114        employees_list_menuitem.addActionListener(al);
115        JMenuItem employees_sort_menuitem = new JMenuItem("Sort");
116        employees_sort_menuitem.addActionListener(al);
117        JMenuItem employees_newHourlyEmployee_menuitem = new JMenuItem("New Hourly...");
118        employees_newHourlyEmployee_menuitem.setActionCommand("NewHourlyEmployee");
119        employees_newHourlyEmployee_menuitem.addActionListener(al);
120        JMenuItem employees_newSalariedEmployee_menuitem = new JMenuItem("New Salaried...");
121        employees_newSalariedEmployee_menuitem.setActionCommand("NewSalariedEmployee");
122        employees_newSalariedEmployee_menuitem.addActionListener(al);
123        JMenuItem employees_editEmployee_menuitem = new JMenuItem("Edit Employee...");
124        employees_editEmployee_menuitem.setActionCommand("EditEmployee");
125        employees_editEmployee_menuitem.addActionListener(al);
126        JMenuItem employees_deleteEmployee_menuitem = new JMenuItem("Delete Employee");
127        employees_deleteEmployee_menuitem.setActionCommand("DeleteEmployee");
128        employees_deleteEmployee_menuitem.addActionListener(al);
129        this.delete_employee_menuitem = employees_deleteEmployee_menuitem;
130        this.edit_employee_menuitem = employees_editEmployee_menuitem;
131        if(current_employee_list == null){
132            this.delete_employee_menuitem.setEnabled(false);
133            this.edit_employee_menuitem.setEnabled(false);
134        }

```

```

135
136     employees_menu.add(employees_list_menuitem);
137     employees_menu.add(employees_sort_menuitem);
138     employees_menu.add(employees_newHourlyEmployee_menuitem);
139     employees_menu.add(employees_newSalariedEmployee_menuitem);
140     employees_menu.add(employees_editEmployee_menuitem);
141     employees_menu.add(employees_deleteEmployee_menuitem);
142
143     menubar.add(employees_menu);
144
145     return menubar;
146 }
147
148 private void setupGUI(ActionListener al){
149     edit_employee_dialog = new EditEmployeeDialog(this);
150     textareal = new JTextArea(initial_textarea_rows, initial_textarea_columns);
151     JScrollPane scrollpane = new JScrollPane(textareal);
152     this.getContentPane().add(scrollpane);
153     this.setJMenuBar(this.setupMenuBar(al));
154     this.setSize(initial_frame_height, initial_frame_width);
155     this.setLocation(initial_frame_position_x, initial_frame_position_y);
156     this.pack();
157     this.show();
158 }
159
160 private int getSelectedLineNumber(){
161     int caret_position = textareal.getCaretPosition();
162     int line_number = 0;
163     int string_length = 0;
164     try{
165         StringTokenizer st = new StringTokenizer(textareal.getText(), "\n");
166         while(caret_position >= (string_length += st.nextToken().length() )){
167             line_number++;
168         }
169     }catch(NoSuchElementException ignored){ }
170
171     if(line_number >= current_employee_list.length){
172         line_number = current_employee_list.length-1;
173     }
174     return line_number;
175 }
176
177 private String[] employeeInfoStringToStringArray(String emp_string){
178     StringTokenizer st = new StringTokenizer(emp_string);
179     String fname = st.nextToken();
180     String mname = st.nextToken();
181     String lname = st.nextToken();
182     String gender = st.nextToken();
183     String bday = st.nextToken();
184     String emp_no = st.nextToken();
185
186     StringTokenizer st2 = new StringTokenizer(bday, "/");
187     String day = st2.nextToken();
188     String month = st2.nextToken();
189     String year = st2.nextToken();
190
191     String[] string_array = {fname, mname, lname, gender, year, month, day, emp_no};
192     return string_array;
193 }
194
195 public File getSaveFile(){
196     File file = null;
197     JFileChooser file_chooser = new JFileChooser();
198     int result = file_chooser.showSaveDialog(this);
199     switch(result){
200         case JFileChooser.APPROVE_OPTION: { file = file_chooser.getSelectedFile();
201             break;
202         }
203         case JFileChooser.CANCEL_OPTION: break;
204     } // end switch
205     return file;
206 }
207
208
209 public File getLoadFile(){
210     File file = null;
211     JFileChooser file_chooser = new JFileChooser();
212     int result = file_chooser.showOpenDialog(this);
213     switch(result){
214         case JFileChooser.APPROVE_OPTION: { file = file_chooser.getSelectedFile();
215             break;

```

```

216         }
217         case JFileChooser.CANCEL_OPTION: break;
218
219     } // end switch
220     return file;
221 }
222 }
223 } // end View class definition

```

25.42 EditEmployeeDialog.java

```

1     package com.pulpfreepress.view;
2
3     import javax.swing.*;
4     import java.awt.*;
5     import java.awt.event.*;
6     import java.util.regex.*;
7
8     public class EditEmployeeDialog extends JDialog implements ActionListener {
9         private JLabel label1 = null;
10        private JLabel label2 = null;
11        private JLabel label3 = null;
12        private JLabel label4 = null;
13        private JLabel label5 = null;
14        private JLabel label6 = null;
15        private JLabel label7 = null;
16        private JLabel label8 = null;
17        private JLabel label9 = null;
18        private JLabel label10 = null;
19        private JLabel label11 = null;
20
21        private JTextField textfield1 = null;
22        private JTextField textfield2 = null;
23        private JTextField textfield3 = null;
24        private JTextField textfield4 = null;
25        private JTextField textfield5 = null;
26        private JTextField textfield6 = null;
27        private JTextField textfield7 = null;
28        private JTextField textfield8 = null;
29        private JTextField textfield9 = null;
30        private JTextField textfield10 = null;
31        private JTextField textfield11 = null;
32
33
34        public EditEmployeeDialog(JFrame frame){
35            super(frame, "Employee Information Dialog", true); // modal dialog
36            this.setupGUI();
37            this.setVisible(false);
38        }
39
40        public void actionPerformed(ActionEvent ae){
41            if(ae.getActionCommand().equals("Submit")){
42                boolean bool_val = verifyFieldValues();
43                if(bool_val){
44                    this.setVisible(false);
45                }
46            }
47        }
48
49        private void setupGUI(){
50            label1 = new JLabel("First Name:");
51            label2 = new JLabel("Middle Name:");
52            label3 = new JLabel("Last Name:");
53            label4 = new JLabel("Gender:");
54            label5 = new JLabel("Birth Year:");
55            label6 = new JLabel("Birth Month:");
56            label7 = new JLabel("Birth Day:");
57            label8 = new JLabel("Employee Number:");
58            label9 = new JLabel("Hours Worked:");
59            label10 = new JLabel("Hourly Rate");
60            label11 = new JLabel("Salary:");
61
62            textfield1 = new JTextField(20);
63            textfield2 = new JTextField(20);
64            textfield3 = new JTextField(20);
65            textfield4 = new JTextField(20);
66            textfield5 = new JTextField(20);
67            textfield6 = new JTextField(20);
68            textfield7 = new JTextField(20);
69            textfield8 = new JTextField(20);
70            textfield9 = new JTextField(20);

```

```

71     textfield10 = new JTextField(20);
72     textfield11 = new JTextField(20);
73
74
75     JButton button1 = new JButton("Submit");
76     button1.addActionListener(this);
77
78     this.getContentPane().setLayout(new GridLayout(12, 2, 0, 0));
79     this.getContentPane().add(label1);
80     this.getContentPane().add(textfield1);
81     this.getContentPane().add(label2);
82     this.getContentPane().add(textfield2);
83     this.getContentPane().add(label3);
84     this.getContentPane().add(textfield3);
85     this.getContentPane().add(label4);
86     this.getContentPane().add(textfield4);
87     this.getContentPane().add(label5);
88     this.getContentPane().add(textfield5);
89     this.getContentPane().add(label6);
90     this.getContentPane().add(textfield6);
91     this.getContentPane().add(label7);
92     this.getContentPane().add(textfield7);
93     this.getContentPane().add(label8);
94     this.getContentPane().add(textfield8);
95     this.getContentPane().add(label9);
96     this.getContentPane().add(textfield9);
97     this.getContentPane().add(label10);
98     this.getContentPane().add(textfield10);
99     this.getContentPane().add(label11);
100    this.getContentPane().add(textfield11);
101    this.getContentPane().add(new JPanel());
102    this.getContentPane().add(new JPanel().add(button1));
103
104    this.pack();
105 }
106
107 public String[] getEmployeeInfo(){
108     String[] emp_info = new String[11];
109
110     emp_info[0] = textfield1.getText();
111     emp_info[1] = textfield2.getText();
112     emp_info[2] = textfield3.getText();
113     emp_info[3] = textfield4.getText();
114     emp_info[4] = textfield5.getText();
115     emp_info[5] = textfield6.getText();
116     emp_info[6] = textfield7.getText();
117     emp_info[7] = textfield8.getText();
118     emp_info[8] = textfield9.getText();
119     emp_info[9] = textfield10.getText();
120     emp_info[10] = textfield11.getText();
121
122     return emp_info;
123 }
124
125
126 public void hideSalaryInfo(){
127     label11.setVisible(false);
128     textfield11.setVisible(false);
129 }
130
131 public void showSalaryInfo(){
132     label11.setVisible(true);
133     textfield11.setVisible(true);
134 }
135
136 public void hideHourlyInfo(){
137     label9.setVisible(false);
138     textfield9.setVisible(false);
139     label10.setVisible(false);
140     textfield10.setVisible(false);
141 }
142
143 public void showHourlyInfo(){
144     label9.setVisible(true);
145     textfield9.setVisible(true);
146     label10.setVisible(true);
147     textfield10.setVisible(true);
148 }
149
150 public void clearTextFields(){
151     textfield1.setText("");

```

```

152         textfield2.setText("");
153         textfield3.setText("");
154         textfield4.setText("");
155         textfield5.setText("yyyy");
156         textfield6.setText("mm");
157         textfield7.setText("dd");
158         textfield8.setText("");
159         textfield9.setText("0.00");
160         textfield10.setText("0.00");
161         textfield11.setText("0.00");
162     }
163
164     public void hideHourlyAndSalaryInfo() {
165         hideSalaryInfo();
166         hideHourlyInfo();
167     }
168
169     public void disableEmployeeNumberTextField() {
170         textfield8.setEnabled(false);
171     }
172
173     public void enableEmployeeNumberTextField() {
174         textfield8.setEnabled(true);
175     }
176
177
178     public void populateTextFields(String[] emp_info) {
179         textfield1.setText(emp_info[0]);
180         textfield2.setText(emp_info[1]);
181         textfield3.setText(emp_info[2]);
182         textfield4.setText(emp_info[3]);
183         textfield5.setText(emp_info[4]);
184         if (Integer.parseInt(emp_info[5]) < 10) {
185             textfield6.setText("0" + emp_info[5]);
186         } else {
187             textfield6.setText(emp_info[5]);
188         }
189         if (Integer.parseInt(emp_info[6]) < 10) {
190             textfield7.setText("0" + emp_info[6]);
191         } else {
192             textfield7.setText(emp_info[6]);
193         }
194         textfield8.setText(emp_info[7]);
195     }
196 }
197
198 public String[] getEditedEmployeeInfo() {
199     String[] emp_info = new String[8];
200
201     emp_info[0] = textfield1.getText();
202     emp_info[1] = textfield2.getText();
203     emp_info[2] = textfield3.getText();
204     emp_info[3] = textfield4.getText();
205     emp_info[4] = textfield5.getText();
206     emp_info[5] = textfield6.getText();
207     emp_info[6] = textfield7.getText();
208     emp_info[7] = textfield8.getText();
209
210
211     return emp_info;
212 }
213
214 private boolean verifyFieldValues() {
215     boolean ok = true;
216
217     if (Pattern.matches("[A-Za-z]{1,64}", textfield1.getText())) {
218         // do nothing
219     } else {
220         textfield1.setText("-----Invalid First Name-----");
221         ok = false;
222     }
223
224     if (Pattern.matches("[A-Za-z]{1,64}", textfield2.getText())) {
225         // do nothing
226     } else {
227         textfield2.setText("-----Invalid Middle Name-----");
228         ok = false;
229     }
230
231     if (Pattern.matches("[A-Za-z]{1,64}", textfield3.getText())) {
232         // do nothing

```

```

233     } else {
234         textfield3.setText("-----Invalid Last Name-----");
235         ok = false;
236     }
237
238
239     if(textfield4.getText().equals("Male") || textfield4.getText().equals("Female")){
240         // do nothing
241     } else {
242         textfield4.setText("-----Male or Female-----");
243         ok = false;
244     }
245
246     if(Pattern.matches("[1|2]{1}[0-9]{3}", textfield5.getText())){
247         // do nothing
248     } else {
249         textfield5.setText("----invalid Birth Year----");
250         ok = false;
251     }
252
253     if(Pattern.matches("[0|1]{1}[0-9]{1}", textfield6.getText())){
254         if((Integer.parseInt(textfield6.getText()) < 1) ||
255            (Integer.parseInt(textfield6.getText()) > 12) ){
256             textfield6.setText("----invalid Birth Month----");
257             ok = false;
258         }
259     } else {
260         textfield6.setText("----invalid Birth Month----");
261         ok = false;
262     }
263 }
264
265     if(Pattern.matches("[0-3]{1}[0-9]{1}", textfield7.getText())){
266         if((Integer.parseInt(textfield7.getText()) < 1) ||
267            (Integer.parseInt(textfield7.getText()) > 31) ){
268             textfield7.setText("----invalid Birth Day----");
269             ok = false;
270         }
271     } else {
272         textfield7.setText("----invalid Birth Day----");
273         ok = false;
274     }
275
276     if(Pattern.matches("[0-9]{6}", textfield8.getText())){
277         // do nothing
278     } else {
279         textfield8.setText("----invalid Employee Number----");
280         ok = false;
281     }
282
283     if(Pattern.matches("[0-9]{1,2}[.]{1}[0-9]{2}", textfield9.getText())){
284         // do nothing
285     } else {
286         textfield9.setText("----invalid Hours Worked----");
287         ok = false;
288     }
289
290     if(Pattern.matches("[0-9]{1,3}[.]{1}[0-9]{2}", textfield10.getText())){
291         // do nothing
292     } else {
293         textfield10.setText("----invalid Hourly Rate----");
294         ok = false;
295     }
296
297     if(Pattern.matches("[0-9]{1,7}[.]{1}[0-9]{2}", textfield11.getText())){
298         // do nothing
299     } else {
300         textfield11.setText("----invalid Salary----");
301         ok = false;
302     }
303
304     return ok;
305 }
306
307 } // end EditEmployeeDialog class definition

```

Figure 25-5 shows several screen-shots of the user interface of this application in action.

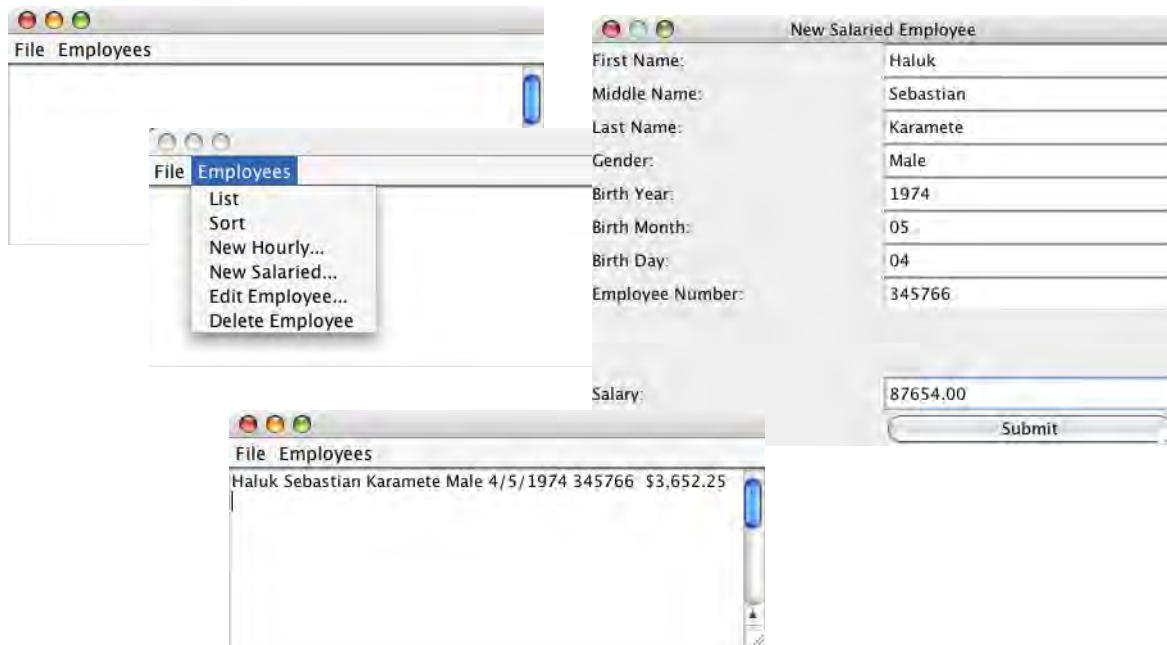


Figure 25-5: Interacting with the Employee Management Application

SUMMARY

Software design patterns are a form of knowledge reuse. Design patterns are general software architectural solutions to general software architectural problems. A design pattern serves as the basis for a specific solution implementation. A complete design pattern specification includes more than just a graphical representation. Some design patterns can be applied alone while others are meant to be combined with other design patterns.

The Singleton pattern is used when only one instance of a particular class type is required to exist in your program. The general approach to creating a Singleton is to make the constructor protected or private and provide a public static method named `getInstance()` that returns the same instance of the class in question.

The Factory pattern is used to create classes whose purpose is to create objects of a specified type. The Dynamic Factory can be used to create objects via Java's dynamic class loading mechanism. One of the primary advantages of the Dynamic Factory pattern is that certain enhancements to an application that uses a dynamic factory can be made and implemented without the need to shut down the application.

The Model-View-Controller (MVC) pattern is used to separate the visual representation of an application object from the application object itself. The MVC pattern consists of three primary components: 1) the *model*, which can consist of one or more classes working together to realize the functionality of a particular application, 2) the *view*, which can consist of one or more classes working together to implement the visual representation of the model, and 3) the *controller*, which can consist of one or more classes working together to coordinate messaging between the model and the view.

The Command pattern is used to 1) decouple the knowledge of a particular action from an object that needs the action carried out, and 2) to encapsulate the action in the form of an object. The Command pattern can be combined with the Dynamic Factory pattern to map command names to class handlers and dynamically load and execute the command handler.

Skill-Building Exercises

1. **Research:** Procure a copy of the Gang-of-Four's Design Patterns book and expand your understanding of object-oriented software design patterns.
2. **UML Sequence Diagram:** Create a UML sequence diagram showing the sequence of events when the button is clicked in the View component of the MVC example given in examples 25.9 through 25.11.
3. **UML Sequence Diagram:** Create a UML sequence diagram showing the sequence of events for command execution when the File->Load menuitem is selected in the Employee Management Application.
4. **Writing Exercise:** Write an essay on the topic of software design patterns. Discuss their history, origin, and utility.
5. **Talk With A Mentor:** Arrange an interview with a senior Java software engineer and ask them how often they use design patterns in their work.
6. **Compile And Run Chapter Examples:** Compile and run the example programs given in this chapter.
7. **Programming:** Modify the CommandProperties class given in example 25.40 so that it checks the date of the properties file before loading it to read the property values. Add a new field to the CommandProperties class called "last_modified" that stores the last modification value of the properties file. Add a new method to the CommandProperties class called loadPropertiesFile() that performs the required last_modified value check and reinitializes the property values as required. Add the capability in the GUI to check for updated property files and to force the reloading of a newly created Command class. (*Note: The new Command class name must differ from the name of Command class it is replacing.*)

Suggested Projects

1. **Legacy Datafile Adapter Revisited:** Write an application with the help of the patterns presented in this chapter that access data via the legacy datafile adapter class and supporting classes presented in chapter 18, examples 18.23 through 18.28. Give the application the capability to list books in the database, add new books, increment and decrement quantity on hand, and search for books by title and author.
2. **Chapter 21 Projects Revisited:** Revisit any of the suggested projects listed at the end of chapter 21 and utilize the patterns discussed in this chapter in their implementation.

Self-Test Questions

1. Why is a software design pattern considered to be a form of knowledge reuse?
2. What is the purpose of the Singleton pattern?
3. What is the purpose of the Factory pattern?
4. What is one potential benefit to using the Dynamic Factory pattern?
5. How can application behavior can be dynamically modified using the Dynamic Factory pattern?

6. What is the purpose of the Model-View-Controller (MVC) pattern?
7. What's the purpose of the Model component of the MVC pattern?
8. What's the purpose of the View component of the MVC pattern?
9. What's the purpose of the Controller pattern of the MVC pattern?
10. Why is it desirable to deny knowledge of the View from the Model and vice versa?
11. What's the purpose of the Command pattern?

REFERENCES

Christopher Alexander. *A Timeless Way of Building*. Oxford University Press, New York. ISBN: 0-19-502402-8

Christopher Alexander, et. al. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York. ISBN: 0-19-501919-9

Erich Gamma, et. al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA. ISBN: 0-201-63361-2

Java 5 Platform API Documentation: [<http://java.sun.com/j2se/1.5.0/docs/api/index.html>]

David Flanagan. *Java™ In A Nutshell*, Fifth Edition. O'Reilly Media, Inc. Sebastopol, CA. ISBN: 0-596-00773-6.

The Hillside Group Website [<http://hillside.net>]

David Stotts, Associate Professor, University of North Carolina Chapel Hill, North Carolina. Patterns information page: [<http://www.cs.unc.edu/~stotts/COMP204/patterns/>]

NOTES

APPENDICES

Appendix A

Helpful CHECKLISTS AND TABLES

PROJECT-APPROACH STRATEGY Check-off List

Check-Off	Strategy Area	Explanation
	Application Requirements	Determine and clarify exactly what purpose and features the finished project must have. Clarify your understanding of the requirements with your instructor if the project specification is not clear. <i>This results in a clear problem definition and a list of required project features.</i>
	Problem Domain	Study the problem until you have a clear understanding of how to solve it. Optionally, express your understanding of the solution by writing a pseudocode algorithm that describes, step-by-step, how you will solve the problem. You may need to do this several times on large, complex projects. <i>This results in a high-level solution statement that can be translated into an application design.</i>
	Language Features	Make a list of all the language features you must understand and use to draft a competent design and later implement your design. As you study each language feature check it off your list. Doing so will give you a sense of progress. <i>This results in a notional understanding of the language features required to effect a good design and solve the problem.</i>
	High-Level Design & Implementation Strategy	Sketch out a rough application design. A design is simply a statement, expressed through words, pictures, or both, of how you plan to implement the problem solution derived in the Problem Domain strategy area. <i>This results in a plan of attack!</i>

Table Appendix A-1: Project Approach Strategy

DEVELOPMENT Cycle

Step	Explanation
Plan	Design to the point where you can get started on the implementation. Do not attempt to design everything up front. The idea here is to keep your design flexible and open to change.
Code	Implement what you have designed.
Test	Thoroughly test each section or module of source code. The idea here is to try to break it before it has a chance to break your application. Even in small projects you will find yourself writing short test-case programs on the side to test something you have just finished coding.
Integrate/Test	Add the tested piece of the application to the rest of the project and then test the whole project to ensure it didn't break existing functionality.
Refactor	This step applies more to object-oriented programming than to procedural programming. It means to take a comprehensive look at your overall application architecture and migrate general functionality up into base, or even abstract, classes so the functionality can be utilized by more concrete derived classes.

Table Appendix A-2: Development Cycle

Appendix B

ASCII Table

ASCII Table

Decimal	Octal	Hex	Binary	Value	Comment
000	000	000	00000000	NUL	Null char
001	001	001	00000001	SOH	Start of Header
002	002	002	00000010	STX	Start of Text
003	003	003	00000011	ETX	End of Text
004	004	004	00000100	EOT	End of Transmission
005	005	005	00000101	ENQ	Enquiry
006	006	006	00000110	ACK	Acknowledgment
007	007	007	00000111	BEL	Bell
008	010	008	00001000	BS	Backspace
009	011	009	00001001	HT	Horizontal Tab
010	012	00A	00001010	LF	Line Feed
011	013	00B	00001011	VT	Vertical Tab
012	014	00C	00001100	FF	Form Feed
013	015	00D	00001101	CR	Carriage Return
014	016	00E	00001110	SO	Shift Out
015	017	00F	00001111	SI	Shift In
016	020	010	00010000	DLE	Data Link Escape
017	021	011	00010001	DC1	XON Device Control 1
018	022	012	00010010	DC2	Device Control 2
019	023	013	00010011	DC3	XOFF Device Control 3
020	024	014	00010100	DC4	Device Control 4
021	025	015	00010101	NAK	Negative Acknowledgement
022	026	016	00010110	SYN	Synchronous Idle
023	027	017	00010111	ETB	End of Trans. Block
024	030	018	00011000	CAN	Cancel
025	031	019	00011001	EM	End of Medium
026	032	01A	00011010	SUB	Substitute
027	033	01B	00011011	ESC	Escape
028	034	01C	00011100	FS	File Separator
029	035	01D	00011101	GS	Group Separator
030	036	01E	00011110	RS	Request to Send Record Separator

Table Appendix B-1: ASCII Table

Decimal	Octal	Hex	Binary	Value	Comment
031	037	01F	00011111	US	Unit Separator
032	040	020	00100000	SP	Space
033	041	021	00100001	!	
034	042	022	00100010	"	
035	043	023	00100011	#	
036	044	024	00100100	\$	
037	045	025	00100101	%	
038	046	026	00100110	&	
039	047	027	00100111	'	
040	050	028	00101000	(
041	051	029	00101001)	
042	052	02A	00101010	*	
043	053	02B	00101011	+	
044	054	02C	00101100	,	
045	055	02D	00101101	-	
046	056	02E	00101110	.	
047	057	02F	00101111	/	
048	060	030	00110000	0	
049	061	031	00110001	1	
050	062	032	00110010	2	
051	063	033	00110011	3	
052	064	034	00110100	4	
053	065	035	00110101	5	
054	066	036	00110110	6	
055	067	037	00110111	7	
056	070	038	00111000	8	
057	071	039	00111001	9	
058	072	03A	00111010	:	
059	073	03B	00111011	;	
060	074	03C	00111100	<	
061	075	03D	00111101	=	
062	076	03E	00111110	>	
063	077	03F	00111111	?	
064	100	040	01000000	@	
065	101	041	01000001	A	
066	102	042	01000010	B	
067	103	043	01000011	C	
068	104	044	01000100	D	
069	105	045	01000101	E	
070	106	046	01000110	F	
071	107	047	01000111	G	
072	110	048	01001000	H	
073	111	049	01001001	I	
074	112	04A	01001010	J	
075	113	04B	01001011	K	
076	114	04C	01001100	L	
077	115	04D	01001101	M	
078	116	04E	01001110	N	
079	117	04F	01001111	O	
080	120	050	01010000	P	
081	121	051	01010001	Q	
082	122	052	01010010	R	
083	123	053	01010011	S	
084	124	054	01010100	T	
085	125	055	01010101	U	
086	126	056	01010110	V	

Table Appendix B-1: ASCII Table

Appendix B

Decimal	Octal	Hex	Binary	Value	Comment
087	127	057	01010111	w	
088	130	058	01011000	x	
089	131	059	01011001	y	
090	132	05A	01011010	z	
091	133	05B	01011011	[
092	134	05C	01011100	\	
093	135	05D	01011101]	
094	136	05E	01011110	^	
095	137	05F	01011111	_	
096	140	060	01100000	`	
097	141	061	01100001	a	
098	142	062	01100010	b	
099	143	063	01100011	c	
100	144	064	01100100	d	
101	145	065	01100101	e	
102	146	066	01100110	f	
103	147	067	01100111	g	
104	150	068	01101000	h	
105	151	069	01101001	i	
106	152	06A	01101010	j	
107	153	06B	01101011	k	
108	154	06C	01101100	l	
109	155	06D	01101101	m	
110	156	06E	01101110	n	
111	157	06F	01101111	o	
112	160	070	01110000	p	
113	161	071	01110001	q	
114	162	072	01110010	r	
115	163	073	01110011	s	
116	164	074	01110100	t	
117	165	075	01110101	u	
118	166	076	01110110	v	
119	167	077	01110111	w	
120	170	078	01111000	x	
121	171	079	01111001	y	
122	172	07A	01111010	z	
123	173	07B	01111011	{	
124	174	07C	01111100		
125	175	07D	01111101	}	
126	176	07E	01111110	~	
127	177	07F	01111111	DEL	

Table Appendix B-1: ASCII Table

INDEX

Symbols

- ! 145
- 141
- 147
- % 143
- %PATH% 32
- & 145, 148
- && 144
- &= 148
- * 141
- *= 148
- + 141, 142
- ++ 147
- += 148
- . 147
- / 141
- /= 148
- < 143, 146
- <<= 148
- <= 143
- <applet> tag 627
- <object> tag 627
- = 148
- == != 144
- > 143
- >= 143
- >> 146
- >>= 148
- >>> 146
- >>>= 148
- ^ 145, 148
- ^= 148
- | 145, 148
- |= 148
- || 144
- ~ 148

A

- abstract 127
 - classes 276
 - methods 276
- abstract class 267, 297
 - expressing in UML 277
 - purpose of 276
 - term defined 267
- abstract class vs. interface 279, 280
- abstract data types 211

- abstract keyword
 - using to declare classes and methods 278
- abstract method 297
- abstract methods
 - implementing in derived classes 278
- abstract thinking 8
- abstraction
 - problem 8
 - the art of programming 210
- abstractions 381
 - assigning properties to 381
 - selecting the right kinds of 741
- access
 - horizontal 283
 - vertical 283
- Access Control Graph (ACG) 687
- access modifier
 - public
 - use of 124
- access modifiers 215
 - behavior of 283
 - default/package 216
 - private 215
 - protected 216
 - public 215
- AccessControlException 631
- ActionEvent 349, 351, 356
- actionPerformed() 612
- ActiveQueue 499
- adapter class
 - to flat-file data file 531
- adapters 374
- address bus 97
- addressing local machine 561
- aggregation 244, 245, 247, 260
 - aggregate constructors 246
 - composite 245, 246, 260
 - composite example code 249
 - definition 245
 - determining type by who controls object lifetime 246
 - effects of JVM garbage collector 246
 - example
 - aircraft engine simulation 251
 - aircraft engine simulation class diagram 252
 - simple 245, 246, 247, 260
 - simple example code 248
 - two types of 688
- algorithm
 - running time 101
 - understanding the concept of 92
 - working definition of 99
- algorithms 92, 99
 - good vs. bad 100
- analysis 66, 680
- anonymous class 349, 372
 - avoiding 17
- anonymous listener 374
- Ansel Adams 680
- ANT
 - website address 244
- Ant 244, 245
- applet 123
 - <applet> tag 630
 - applet tag 628
 - basic example 627
 - connecting to server 631
 - defined 626
 - destroy() 629, 630
 - extended example code 636–638
 - HTML page requirement 628
 - inheritance hierarchy 627
 - init() 629, 630
 - jar file loading explained 630
 - life cycle stages 629
 - milestones 629
 - packaging and distribution 628
 - parameters 634, 635
 - running with browser 628
 - security policy and signed applets 634
 - security restrictions 631, 634
 - start() 629, 630
 - stop() 629, 630
- applet methods
 - four primary 628
- applet objects 123
- applets 626, 626–638
 - benefits of using 626
- application
 - compiling and executing 125
 - graceful recovery 64
 - java class definition
 - structure of 124

- layers 558
 - physical deployment 558
 - physical tier distribution 563
 - tier responsibilities 563
 - tiers 558
 - application class definition structure 124
 - application class structure
 - example of 130
 - application distribution 558
 - across multiple computers 561
 - in single jar file 388
 - application object 123
 - application tiers
 - logical 562
 - separation of concerns 563
 - applications
 - building bigger 126
 - multi-tiered 562
 - architecture
 - flexibility 681
 - modularity 681
 - reliability 681
 - stability 681
 - array
 - converting String elements to ints 134
 - creating with literal values 186
 - declaration syntax 181
 - definition of 180
 - difference between primitive type and reference type arrays 187
 - elements 180
 - functionality provided by array types 182
 - homogeneous elements 180
 - main() method String parameter 200
 - multidimensional 194, 198
 - creating with array literals 196
 - declaration syntax 194
 - of primitive types 183
 - primitive type
 - memory arrangement 184
 - processing 134
 - properties of 183
 - ragged 197
 - references
 - calling Object and Class methods on 185
 - single dimensional 183
 - single dimensional in action 187
 - specifying length 181
 - specifying types 181
 - two dimensional
 - example program 198
 - memory representation of 196
 - visualization of 196
 - type inheritance hierarchy 182
 - array processing 64
 - array types 130
 - arrays 180
 - sorting with Arrays class 201
 - using to solve problems 180
 - Arrays class 201
 - arrays of arrays 194
 - Art of Illusion 398
 - ASCII 29
 - Ashmore's hash code algorithm 706
 - assert 127
 - assert mechanism 231
 - association 245, 260
 - definition 245
 - attribute candidates 65
 - attributes 63
 - autoboxing
 - example 501
 - AWT 305, 350
 - AWT and Swing
 - component naming conventions 307
 - historical background 306
 - package names 307
 - AWTEvent 356, 371
- B**
- bad software architecture
 - characteristics of 740
 - base class 266, 297
 - methods
 - overriding 275
 - source code example 269
 - BaseCommand class 754
 - behavior
 - generalized 266
 - Bertrand Meyer 683, 736
 - Bertrand Meyer's Design by Contract (DbC) 723
 - bit 96, 97
 - Bloch's equals() Method Criterion 704
 - Bloch's hash code algorithm 706
 - blocked
 - thread 456
 - blocking
 - accept() method 584
 - BlockingQueue 500
 - Boolean 149
 - boolean 127
 - BorderLayout 321
 - EAST 321
 - NORTH 321
 - SOUTH 321
 - WEST 321
 - bounds
 - of GUI components 308
 - break 127, 159
 - bridge 555
 - Browser JVM 627
 - BufferedOutputStream
 - buffer behavior 517
 - flushing the buffer 517
 - setting buffer size 517
 - Byte 149
 - byte 96, 97
- C**
- cache memory 96
 - Canvas class
 - extending 595
 - Cartesian coordinate 307
 - case 127, 160
 - casting 274
 - advice on use of 275
 - catch 127
 - ChangeEvent 349, 351, 371
 - char 127
 - Character 149
 - character stream file I/O 524
 - CheckboxListCell 399
 - Christopher Alexander 746
 - class 122, 127, 267, 297
 - abstract 276
 - expressing in UML 277
 - purpose of 276
 - abstract class 267
 - final 285
 - four categories of members 214
 - non-static fields 215
 - non-static methods 215
 - static fields 214
 - static methods 215
 - term definition 122, 267
 - class constant
 - definition of 137
 - class declarations
 - viewed as behavior specifications 724
 - class definition

- accessor methods 222
 - adding fields 221
 - adding instance methods 221
 - constructor method 222
 - getter methods 222
 - setter methods 226
 - starting 221
 - class fields
 - accessing from main() method 136
 - class file 122
 - class files 28
 - class invariant 724, 726
 - defined 724
 - class invariants 724
 - class methods 15
 - class responsibility assignment 590
 - class variable
 - definition of 137
 - Class.forName() 581, 750
 - use of 618
 - classes
 - number in an application 244
 - CLASSPATH 27
 - client 554, 557
 - application 554, 557
 - hardware 554, 557
 - client-server applications 581, 582
 - client-server protocol 607
 - client-side RMI runtime 570
 - clone() method
 - example code 709
 - CloneNotSupportedException 708
 - Coad's Inheritance Criteria 684
 - code blocks
 - executing in if statements 157
 - code reuse 680
 - coding convention
 - adopting 20
 - cohesion 14
 - collateral roles
 - modeling 686
 - Collection 487
 - collections
 - algorithms 492
 - ArrayList
 - usage example 484
 - Arrays class 493
 - casting retrieved objects 496
 - Collections class 492
 - core interfaces 486
 - creating new data structures
 - from existing 497
 - enhanced for loop
 - when to use 501
 - framework
 - organization 486
 - purpose of 486
 - general characteristics 482
 - general purpose implementation
 - classes 487
 - generic methods 501
 - generics 499
 - infrastructure 493
 - interfaces 482
 - Java 1.4.2 style 493–499
 - Java 5 core interfaces 499
 - Java 5 style 499–502
 - linked list node elements 489
 - performance characteristics
 - arrays 488
 - hashtable 491
 - linked list 489
 - red-black tree 491
 - red-black tree node elements 492
 - Set class
 - example usage of 494
- command line arguments
 - using in program 133
- Command pattern 746, 753
- CommandFactory class 755
- comments
 - javadoc 19
 - multi-line 18
 - single-line 18
 - three Java types 18
- Comparable interface 711
- Comparator interface
 - implementing 713
- compiler errors
 - fixing 12
- compiling multiple source files 244
- complex application behavior 244
- complexity
 - conceptual 12, 244, 245, 260
 - managing physical 14
 - physical 13, 244, 245, 260
 - relationship between physical and conceptual 14
- Component
 - paint() 384
- component 308
 - absolute positioning 316
 - when to paint() 384
- component bounds 308
- ComponentEvent 359
- components 329–337
 - drawing onto 384
- methods for setting renderers 393
- organizing into containers 316
- composite aggregation
 - defined 245
- composition 680, 688
 - as force multiplier 688
- compositional design 244, 688
- compositionists 680
- computer
 - architecture
 - feature set 95
 - feature set accessibility 95
 - feature set implementation 95
 - three aspects of 95
 - definition of 92
 - memory
 - organization 95
 - processing cycle 98
 - system 92
 - components of 93
 - hard drive 93
 - keyboard 93
 - main logic board 93
 - memory 93, 96
 - monitor 93
 - mouse 93
 - processor 93, 94
 - speakers 93
 - system unit 93
 - vs. computer system 92
- computer network
 - definition 554
 - purpose 554
- computer program
 - modeling real-world problem 210
- computer screen coordinate system 305
- computers 92
- conceptual complexity 12, 244
 - managing 12
 - taming 13
- concurrently
 - performing activities 444
- ConcurrentMap 500
- configuration-management tool 14
- connection
 - incoming client 582
- const 128
- constant 65
 - declaring in main() method 132
 - definition of 132
 - usage in main() method 132
- constructor
 - purpose of 135

constructor call 135
 constructor method 15
 consumer thread 468
 container 308
 containers
 top-level 308
 containing aggregate 247
 containment
 by reference 246
 by value 246
 polymorphic 688
 contains 247
 continue 128
 control bus 97
 controller 751
 coordinates 383
 coupling 14
 creativity
 and problem abstraction 210
 current position 66
 custom editor
 example use of 412
 custom exceptions
 creating 438
 custom painting 385
 custom renderer 379
 example use of 412
 writing 392

D

data bus 97
 data encapsulation 235
 violating 235
 data type 65
 primitive 182
 reference 182
 data types
 array 182
 primitive
 boolean 129
 byte 129
 char 129
 double 130
 float 130
 int 130
 long 130
 short 129
 working with 130
 reference 130
 DataInputStream 610
 DataInputStream class 581
 DataInputStream.readUTF() 584
 DataOutputStream 610, 612
 DataOutputStream class 581

DbC 723
 deadlock 474
 deep copy
 defined 708
 default 128
 Department of Defense 556
 dependencies
 managed 681
 dependency 245
 definition 245
 effects of dependency relationships between classes 245
 dependency inversion principle 740
 dependency relationship 260
 dependency vs. association 245
 derived class 266, 297
 source code example 270
 derived requirements 590
 design 680
 design by composition 244
 Design by Contract 723
 design pragmatists 680
 development cycle 61
 applying 61
 code 61, 782
 deploying 61
 factor 61
 integrate 61
 plan 61, 782
 test 61, 782
 using 61
 dialog 309
 modality 309
 Dialogue with a Skeptic 313
 difference between abstract class and interface 279
 direction 65
 distributed applications 554
 do 128
 dominant roles
 modeling 686
 Double 149
 double 128
 double buffered 385
 double buffering 385
 Dr. Barbara Liskov 723
 Dr. Bertrand Meyer 723
 DragList 416
 DressingBoard class 382
 dynamic class loading 750
 example code 755
 Dynamic Factory Pattern
 advantages of 751

dynamic polymorphic behavior 266
 DynamicArray
 case study 482

E

Eclipse 30
 editor
 writing custom 408
 effects of change
 predicting 681
 Eiffel 723
 else 128
 Emeril Lagasse 288
 empty statement
 example 138
 enabling assert in javac 231
 enabling assertions in Java VM 231
 enclosing class 366
 engineering trade-off 680
 enhanced for loop 499, 501
 entity diagram 650
 enum 128
 enumeration
 typesafe 254
 environment variables
 CLASSPATH 31
 PATH 31
 purpose of CLASS & CLASSPATH 32
 setting 27
 setting in Windows 31
 equals() method
 equivalence relation 703
 example code 704
 testing 705
 Erich Gamma 747
 error checking 64
 Error class 429
 errors
 compiler 12
 correcting in source file 29
 Ethernet 565
 Event 350
 event
 custom code example 290
 event listener interfaces 349
 Event listeners 351
 event listeners
 registering with components 351
 registration methods 355
 event registering methods 352
 event-handling framework 350
 EventListeners

- defining 403
 - EventObject 350, 351, 371
 - Events
 - defining 403
 - events
 - choosing the right type 352
 - handling GUI 350
 - low level 349
 - naming conventions 351
 - semantic 349
 - exceptions 428
 - checked 430
 - checked vs. unchecked 430
 - custom
 - creating 438
 - Error class 429
 - Exception class 429
 - extending 438
 - handling 430
 - low vs. high level 438
 - low-level to high-level translation 534
 - multiple try/catch blocks 431
 - ordering
 - importance of 432
 - purpose of 428
 - RuntimeException class 429
 - throwing 436
 - throws clause 436
 - try/catch block 430
 - try/catch/finally block 435
 - try/finally block 435
 - unchecked 430
 - expression 138
 - extends 128
 - extends keyword
 - using to create derived classes 278
 - extension inheritance
 - complications from using 688
 - vs. functional variation 688
- F**
- Façade 746
 - Factory 746
 - factory class 607
 - interfaces involved to employ 686
 - factory class pattern
 - in action 602
 - factory pattern
 - example code 692
 - false 129
 - faux-composite component 393
 - File class 508
 - using 513
 - file I/O 508
 - ASCII 509
 - buffer
 - flushing 511
 - BufferedInputStream 521
 - BufferedOutputStream 516
 - BufferedReader 528
 - BufferedWriter 525
 - buffering
 - purpose of 511
 - write operations 511
 - byte streams 515
 - categorizing I/O classes 510
 - character streams 524
 - choosing the right class 510
 - DataInputStream 522
 - DataOutputStream 517
 - File class
 - using 513
 - file terminal classes 511
 - FileInputStream 521
 - FileOutputStream 515
 - FileReader 527
 - FileWriter 524
 - fixed-length-record
 - locking with lock_token 534
 - InputStreamReader 527
 - InputStreams 520
 - intermediate classes 511
 - Object
 - calling notifyAll() method 534
 - calling wait() method 534
 - ObjectInputStream 523
 - ObjectOutputStream 518
 - OutputStreamWriter 526
 - PrintStream 520
 - PrintWriter 526
 - Properties class
 - example of use 529
 - RandomAccessFile class
 - use of 531–544
 - Reader classes 527
 - sink
 - term defined 508
 - source
 - term defined 508
 - synchronized keyword usage 534
 - Unicode characters 509
 - user fronting classes 511
 - UTF-8 encoding 509
 - files
 - reading bytes into byte array 521
 - reading contents into character array 527
 - final 128
 - final class 285
 - example of 714
 - final method 285
 - final variable 132
 - finalize method 581
 - finalize()
 - use of 618
 - finalize() method
 - alternative approach 710
 - time-sensitive issues 710
 - finally 128
 - Float 149
 - float 128
 - floor 66
 - flow 10
 - achieving 10
 - concept of 10
 - stages 10
 - FlowLayout 318
 - alignment options 318
 - for 128
 - for each loop 501
 - for-each loop 501
 - frame 309
 - fundamental language features 64
- G**
- garbage collection 136
 - garbage collector 618
 - Garment class 381
 - gate 746
 - gateway 555
 - generalization
 - expressing in UML 268
 - generalized behavior
 - specifying 266
 - generic collection classes 501
 - generic method 501
 - generic methods 501
 - generics 499
 - good design
 - goals of 681
 - good software architecture
 - characteristics of 741
 - goto 128
 - Graphics 383, 421
 - offscreen 384
 - onscreen 384
 - performing drawing operations on 384

- graphics
 - property related methods 383
- graphics abilities
 - of Java API 382
- Graphics class
 - function of 383
- graphics drawing operations
 - examples of 383
- graphics method
 - three categories of 383
- Graphics2D 386, 421
- GridBagConstraints 323, 324
 - properties of 324
- GridLayout 323
 - GridBagConstraints 323
- GridLayout 320
- GUI
 - mathematics 307
 - screen coordinate systems 307
- GUI events 354–375
 - handling 350
 - external class with fields 362
 - with anonymous class 372
 - with external class 358, 369
 - with inner class (field level) 366
 - with inner class (local-variable level) 368
- GUI programming 306–347

H

- hardest thing about learning to program 4
- has a 247
- hash code
 - algorithm 706
- hashCode() method
 - example code 707
 - general contract 706
 - purpose of 705
 - testing 707
- HashMap
 - used to store properties 530
- homogeneous data types 180
- horizontal access 216, 283
- host 557, 567

I

- I/O
 - file 508
 - java.io package overview 508
- IDE 27, 30
- identifier

- class name examples 21
- constant name examples 21
- correct formulation of 126
- method name examples 22
- naming 19
 - example of 126
- naming rules 126
- testing for validity 127
- unicode 127
- variable name examples 21
- well-named 127
- if 128
- if/else statement 158
- images 383
 - drawing an array of 382
 - loading 387
 - loading with Toolkit class example code 593
- immutable properties 381
- implements 128
- import 128
- incoming client connections 582
- inheritance 680, 682–686
 - first purpose of 266
 - good reasons for using 682
 - Meyer’s Taxonomy 683
 - object-oriented programming with 266
 - second purpose of 267
 - simple example 269
 - third purpose of 267
 - three purposes of 266
 - valid usage checkpoints 684
- inheritance form
 - constant 684
 - extension 683
 - facility 684
 - functional variation 684
 - implementation 684
 - machine 684
 - model 683
 - reification 684
 - restriction 683
 - software 684
 - structure 684
 - subtype 683
 - type variation 684
 - uneffecting inheritance 684
 - variation 684
 - view 684
- inheritance hierarchy
 - assessing with Coad’s criteria 685
- inheritists 680

- inner class 349, 366
 - avoiding 17
- inner classes
 - generated class files 368
- input
 - simple
 - getting into program 133
- InputStream 610
- InputStream class 508
- InputStreams 508
- instance constant
 - definition of 137
- instance methods 15
- instance variable
 - definition of 137
- instanceof 128
- int 128
- Integer 149
- integrated development environment (IDE) 27
- integrated development environments 30
- interface 128, 266, 267, 297
 - authorized members 279
 - purpose of 279
 - reducing dependencies with 686
 - role of 686
 - term definition 267
- interfaces 680
 - expressing in UML 280
 - implementing 266
- internationalization
 - with Readers and Writers 524
- Internet Protocol (IP) 565
- Internet protocol layers 564
- Internet Protocols 556
- invoking remote methods 569
- IOStreams
 - use in network programming 554
- IP 565
- is a relationship
 - implementing 267
- Iterator 487, 499, 501
 - example use 494
 - next() 494
 - purpose of 501
 - use of 496

J

- J2SDK 31
 - downloading 31
 - installing 31
 - selecting installation directory 31

- jar 27
 - jar files
 - creating from complex package structure 668
 - for applets 630
 - Java
 - API
 - class inheritance hierarchy navigating 112
 - deprecated methods 115
 - documentation 111
 - obtaining information about 111
 - API package diagram 110
 - application
 - two meanings 123
 - applications
 - talking about 123
 - writing 123
 - class construct 214
 - HotSpot Virtual Machine 110
 - JRE 110
 - platform 110
 - SDK 110
 - SDK vs. JRE 110
 - type categories 129
 - java 27, 28
 - using 125
 - Java 5 collections 482
 - Java API
 - support for unanticipated uses 399
 - Java Beans 223
 - Java class structure
 - example 14
 - Java Collections Framework 482–503
 - purpose of 486
 - Java compiler tool
 - using 125
 - Java Database Connectivity 626
 - Java HotSpot Virtual Machine 102
 - architecture 103
 - client 102
 - obtaining 102
 - server 102
 - Java platform
 - continuous evolution effects of 482
 - Java Plug-In 627
 - Java Project 27
 - steps to creating 27
 - java project
 - steps to creating 28
 - Java RMI runtime environment 569
 - Java source file structure 14
 - Java source files
 - general rules for creating 16
 - java source files 28
 - Java virtual machine
 - running programs with 125
 - java.awt.Graphics2D 382
 - java.rmi.Remote interface 599
 - java.util.EventListener 351
 - javac 16, 27, 28
 - assertions
 - enabling 726
 - compiling entire source directory 245
 - compiling multiple source files 244
 - using 125
 - javadoc 27
 - embedding HTML within 19
 - generating HTML API pages 19
 - JBuilder 30
 - JButton 331
 - JCheckBox 331
 - JColorChooser 330
 - JComboBox 330, 369, 408
 - JDBC 626, 639–672
 - 3-tiered application architecture 641
 - driver classes 640
 - packages described 639
 - PreparedStatement
 - recommended usage of 672
 - project description 641–643
 - purpose of 639
 - ResultSetMetaData 655
 - specification 639
 - SQL 640
 - Statement 672
 - steps to employ 641
 - using in project with applets and RMI 641
 - What you need to know to use... 640
 - JDialog 309, 330
 - Jeremy Gibbons 477
 - JFrame 309, 330
 - JLabel 331
 - JList 330, 373, 389, 400
 - JMenu 330
 - JMenuBar 330
 - JMenuItem 331
 - John Vlissides 747
 - JOptionPane 330
 - JPanel 330
 - JPasswordField 331
 - JProgressBar 371
 - JRadioButton 331
 - JRadioButtons
 - five button types 369
 - JRMP
 - version 1.1 vs. 1.2 570
 - JScrollPane 330
 - JSlider 330, 371
 - JSpinner 371
 - JTabbedPane 371
 - JTable 373, 408
 - JTextArea 331
 - JTextField 331
 - JToggleButton 331, 369
 - JToolTip 331
 - JTree 408
 - JViewport 371
 - JVM
 - running multiple on one machine 558
 - starting multiple in Linux 559
 - starting multiple in UNIX 559
 - starting multiple in Windows 558
 - JVMs
 - running multiple 562
 - JWindow 309, 330
- ## K
- keyboard focus 364
 - KeyEvent 349, 351, 364
 - keywords
 - reserved 127
 - killing UNIX processes 560
- ## L
- language features 60, 781
 - layout manager 317
 - BorderLayout 321
 - constraints 321
 - FlowLayout 318
 - GridBagLayout 323
 - GridLayout 320
 - layout managers 305
 - combining with JPanels 327
 - LayoutManager2 321
 - LinkedList 494, 499, 501
 - Liskov Substitution Principle
 - relationship to Meyer Design by Contract Programming 723
 - three rules of 735
 - Liskov Substitution Principle (LSP)

- 723
 - List 487
 - listeners
 - linking to events 354
 - ListeningMainFrame 381
 - listing UNIX processes 560
 - ListIterator 487
 - ListModel 389
 - interface methods 390
 - using 389
 - ListSelectionEvent 349, 351
 - Local Area Network 554
 - localhost 561
 - localhost address 586
 - lock
 - Object 460
 - Long 149
 - long 128
 - low-level events 349
 - LSP 723
 - LSP & DbC
 - common goals 723
 - designing with 724
 - Java support for 723
- M**
- machine code 95
 - Macintosh OS X
 - developer tools 27
 - magic values
 - eliminating the need for 613
 - main application
 - creating separate file 14
 - main application class file
 - creating 17
 - main() method 124
 - body 124
 - different forms of 124
 - parameter 124
 - main() method String array
 - using 133
 - Map 487
 - memory
 - address bus 97
 - alignment 97
 - bit 96, 97
 - byte 96, 97
 - cache 96
 - control bus 97
 - data bus 97
 - hierarchy 96
 - non-volatile 96
 - organization 95
 - RAM 96
 - ROM 96
 - volatile 96
 - word 96, 97
 - menu 65
 - metadata 655
 - method
 - cohesion 217
 - definition structure 217
 - final 285
 - method stubbing 12
 - methods 64, 217
 - abstract 276
 - body 219
 - example definitions 219
 - local variable scoping 233
 - modifiers 218
 - name 219
 - naming 217
 - overloading 220
 - parameter behavior 232
 - parameter list 219
 - passing arguments to 232
 - return types 219
 - signatures 220
 - synchronizing 466
 - using return values as arguments 233
 - methods rule 735
 - modal 309
 - model 63, 751
 - modeling 63
 - collateral roles 687
 - dominant roles 687
 - dynamic roles 687
 - Model-View-Controller 746
 - Model-View-Controller (MVC) 751
 - modulus operator
 - example use 494
 - monitor 460
 - mouse events
 - handling 361
 - MouseEvent 349, 351, 359, 361
 - MouseListener 361, 400
 - MouseMotionListener 361
 - MouseWheelListener 361
 - multiple JVMs 562
 - multi-threaded 588
 - term defined 567
 - multi-threaded server
 - example 610
 - multi-threaded server applications 582
 - multi-threading 444
 - multi-tiered applications 554, 562
 - mutable properties 382
 - MVC 751, 753
 - and ActionListener interface 753
 - Controller
 - using Factory pattern 754
 - simple example of 752
 - MySQL 626, 641, 643–656
 - access control tables 647
 - column permissions 649
 - configuration 643
 - creating databases 647
 - creating DB permissions 648
 - creating tables 650
 - creating tables with SQL script 650
 - db table 648
 - establishing database security 647–650
 - executing SQL script 650, 651
 - helpful commands 646
 - host table 649
 - how to enter SQL commands at monitor prompt 645
 - inserting data into tables 651
 - joining tables 652
 - mysql monitor program 645
 - mysqladmin program 644
 - obtaining and installing 644
 - overview 643
 - populating table data 651
 - security strategy 649
 - select statement 652
 - showing tables and table structure 645
 - steps required to run examples 643
 - table permissions 648
 - updating table data 652
 - user table 647
 - MySQL driver class 655
- N**
- Naming.bind()
 - use of 572, 598
 - native 128
 - NetBeans 30
 - network
 - definition 554
 - homogeneous vs. heterogeneous 555
 - purpose 554
 - network application

- layers 558
- physical deployment 558
- tiers 558
- network applications 554
- network clients
 - running multiple on same machine 561
- network programming
 - client-server scenario 567
 - java support for 567
- networking 554
- networking protocols
 - role of 555
- new 128
- new operator 134
- noun 65
- noun lists
 - suggesting possible application objects 64
- nouns 64, 65
 - mapping to data structures 65
- null 129
- NumberFormat class
 - example usage 191

O

- Object
 - lock 460
- object 122
 - applet 123
 - application 123
 - creating with new operator 135
 - definition of 135
 - object memory address 135
 - term definition 122
 - their associated type 267
- object attributes 64
- object usage scenario evaluation checklist 700
- ObjectInputStream 582
- object-oriented analysis 680
- object-oriented architecture
 - extending 722
 - preferred characteristics 722
 - reasoning about 722
 - understanding 722
- object-oriented design approach 7
- object-oriented programming 210
- object-oriented programming enablers 680
- ObjectOutputStream 582
- objects
 - natural ordering 711
 - operations upon 267
 - proper behavior of 700
 - seven usage scenarios 700
 - well-behaved 700
- OCP 736
 - defined 736
 - example 736
- offscreen graphics 384, 413
- offscreen image 384
 - creating 379
- Ogden Nash 635
- onscreen graphics 384
- open-closed principle 735
 - achieving 736
- operating system commands
 - UNIX and MS-DOS 56
- operator
 - bitwise
 - OR 148
 - XOR 148
 - bitwise AND 148
 - boolean AND 145
 - boolean OR 145
 - boolean XOR 145
 - combination assignment 148
 - complement 148
 - conditional AND 144
 - conditional OR 144
 - equality 144
 - greater than 143
 - instanceof 147
 - left shift 146
 - less than 143
 - member access 147
 - modulus 143
 - new
 - usage of 134
 - right shift 146
 - String concatenation 142
 - ternary conditional 145
 - unary postfix increment 147
 - unary prefix decrement 147
 - unary prefix increment 147
- operator precedence 139
 - using parentheses to force 141
- operators 139
 - arithmetic 141
- OutputStream 610
- OutputStream class 510
- OutputStreams 508
- overriding
 - clone() method 708
 - equals() method 703
 - finalize() method 710

- hashCode() method 703, 705
- toString() method 703

P

- package 128
- packages 17
 - compiling code in complex package structures 667
 - complex package structure example 656
 - importing 380
 - naming 17
 - using to organize code 380
- packet 556
- paint() method 385
 - overriding 594
- paintChildren() method 385
- paintComponent() method 385
- painting 382
- part objects 245, 246
- PATH 27
- PATH environment variable
 - fully-qualified path name 33
- pattern
 - singleton 581
- patterns
 - command 746
 - façade 746
 - factory 686, 746
 - factory class 581
 - MVC 746
 - singleton 686, 746
 - typesafe enumeration 254
- pen 65
- Peter Coad 684
- physical complexity 13, 244
- Pi
 - computing with thread 452
- pixel coordinates 307
- polygons 383
- polymorphic behavior 266
 - example of 276
- polymorphic code 485
- polymorphic containment 688
- polymorphic substitution 686
- polymorphism 680
 - applied 687
 - defined 285, 687
 - goal of programming with 687
 - planning for proper use of 687
- port 567
- postcondition 726
 - defined 725

- postconditions 724
 - changing in derived class methods 731
 - PowerPC 94
 - precondition 726
 - defined 724
 - preconditions 724
 - changing preconditions of derived class methods 728
 - weakening 728
 - primitive type wrapper classes 149
 - primitive types 129
 - private 128, 283
 - problem abstraction 8, 210
 - and the development cycle 210
 - end result of 211
 - mantra 210
 - performing problem analysis 211
 - process of 210
 - problem domain 6, 60, 64, 781
 - procedural-based design approach 7
 - processing cycle 98
 - decode 98, 99
 - execute 98, 99
 - fetch 98, 99
 - store 98, 99
 - processor 94
 - block diagram 94
 - CISC 94
 - machine code 95
 - RISC 94
 - producer thread 468
 - producer-consumer relationship 468
 - production coders vs. design theorists 681
 - program
 - computer perspective 98
 - definition of 98
 - human perspective 98
 - java
 - definition of 123
 - simple
 - skills required to create 123
 - two views of 98
 - program control flow statements 156
 - programming 4
 - basic concepts 122
 - challenges & frustrations 4
 - skills required 4
 - programming as art 4
 - programming cycle 11
 - code 11
 - integrate 11
 - plan 11
 - refactor 11
 - repeating 12
 - summarized 12
 - test 11
 - programs 92
 - creating simple 123
 - why they crash 99
 - project
 - stages
 - automating 30
 - main application execution 29
 - source code compilation 29
 - source file creation 29
 - project approach strategy 6
 - application requirements 6
 - design 7
 - in a nutshell 8
 - language features 7
 - problem domain 6
 - strategy areas 6
 - project complexity
 - managing 12
 - project objectives 63
 - project requirements 6
 - project specification 65
 - projects
 - creating
 - steps to use the J2SDK 31
 - creating in Windows 2000 31
 - properties
 - immutable 381
 - mutable 382
 - Properties class
 - use of
 - extended example 613–619
 - properties rule 735
 - property keys
 - use of 618
 - protected 128, 283
 - protocols
 - custom client-server 588
 - proprietary application 581
 - ps command 560
 - public 128, 283
 - access modifier
 - use of 124
- ## Q
- quality without a name 746
 - Queue 500
 - creating with linked list 497
 - QueueListenerInterface 499
 - QWAN 746
- ## R
- Rabinowitz and Wagon 477
 - race condition 460
 - race conditions 459
 - ragged arrays 197
 - Ralph Johnson 747
 - RandomAccessFile 508
 - RandomAccessFile class 510
 - Reader class 510
 - Readers 508
 - realization 280
 - expanded form 280
 - expressing in UML 280
 - lollipop diagram 280
 - simple form 280
 - record locking 534
 - reference data types 130
 - arrays 130
 - reference to object combinations 271
 - reference types 129
 - working with 134
 - reference variable
 - definition of 135
 - registry
 - starting externally 573
 - relational databases 626
 - creating tables 650
 - foreign key 650
 - primary key 650
 - relating tables 650
 - reliable object-oriented software
 - creating 723
 - Remote interface
 - extending
 - example code 597
 - use of 570
 - Remote Method Invocation 581
 - Remote Method Invocation (RMI) 569–576
 - renderer
 - custom
 - plugin 379
 - repaint() 595
 - repaint() method 385
 - repainting
 - GUI components 382
 - requirements 6, 60, 781
 - gaining insight through pictures 65
 - requirements gathering 6
 - reserved keywords 127
 - resources
 - loading
 - absolute vs. relative URLs 388

- loading via relative URLs 388
 - loading via URL 387
 - return 128
 - Richard Helm 747
 - RMI 590, 596
 - client
 - running 574
 - client application 573
 - client code example 600
 - differences between Java 1.4.2 and Java 1.5 575
 - extended example 596–619
 - extending Remote interface 570
 - extending UnicastRemoteObject 571
 - Java Remote Method Protocol (JRMP) 570
 - Naming.lookup()
 - use of 573
 - overview 569
 - registry
 - starting externally 573
 - starting programmatically 572
 - server
 - running 574
 - server application 572
 - steps to creating application 570
 - RMI registry
 - default port 572
 - starting from program 598
 - RMI runtime 599, 607, 618
 - RMI server-side objects 607
 - rmic tool
 - example use 599, 667
 - use of 571
 - Robert's Rules of Order 555
 - robot rat project specification 62
 - analyzing 63
 - RobotRat project specification 588
 - rule-of-thumb
 - one class-one file 17
- S**
- screen coordinates 308
 - origin 307
 - x axis 308
 - y axis 308
 - selection statements 156
 - self-commenting code
 - writing 19
 - semantic events 349
 - separable model architecture 389
 - Serializable interface 716
 - server 554, 557
 - application 554, 557
 - hardware 554, 557
 - treated as capital equipment 557
 - ServerSocket 567, 582
 - example use of 584
 - ServerSocket class 581
 - ServerSocket.accept() 584
 - Set 487
 - setter methods 226
 - shallow copy
 - defined 708
 - shallow vs. deep copy 708
 - Short 149
 - short 128
 - signature rule 735
 - simple aggregation
 - defined 245
 - simple programs
 - writing 122
 - simple vs. composite aggregation 246
 - SimpleServer application 583
 - simplification
 - of real-world problems 210
 - Singleton 746
 - Singleton pattern 581, 614, 615
 - Socket 567, 582, 610, 612
 - passing to separate thread 567
 - Socket class 581
 - Socket.close() 583
 - Socket.getInputStream() 584
 - Socket.getOutputStream() 584
 - socket-based client
 - extended example 611
 - socket-based client example 585
 - socket-based client-server 582
 - socket-based client-server connection scenario 582
 - socket-based server
 - extended example 607–619
 - software design 212
 - software design patterns 746
 - Abstract Factory 749
 - background 746
 - Command 753
 - definition 746
 - Dynamic Factory 750
 - Factory 749
 - Factory Method 749
 - Singleton 748
 - specification template 747
 - software development roles 5
 - analyst 5
 - architect 5
 - programmer 6
 - SortedMap 487
 - SortedSet 487
 - sorting
 - arrays with Arrays class 201
 - source files
 - compiling 28
 - saving with .java extension 29
 - specialization
 - expressing in UML 268
 - spigot algorithm
 - for generating Pi 477
 - SQL 640
 - executing via JDBC 654
 - insert statement 651
 - joining tables 652
 - nested select statement example 654
 - running script in MySQL 651
 - select statement 652
 - clauses 652
 - update statement 652
 - statement 138
 - empty 138
 - for
 - personality of 165
 - statements
 - break 168
 - chained if/else 159
 - continue 168
 - control flow 156
 - do/while 164
 - personality of 164
 - executing consecutive if 157
 - for 165
 - relationship to while 165
 - if 156
 - if/else 156, 158
 - iteration 162
 - nesting 166
 - labeled break 169
 - labeled continue 170
 - mixing selection and iteration 166
 - selection statements 156
 - switch 156, 159
 - falling through cases 160
 - nested 161
 - using break in 159
 - table of 171
 - unlabeled break 168
 - unlabeled continue 170
 - while 162
 - personality of 163

- static 128
 - static initializer 595
 - example code 594
 - static initializers 234
 - static method
 - main() 137
 - static polymorphism 501
 - strategy
 - project approach 6
 - strengthening preconditions 730
 - strictfp 129
 - String
 - array of 190
 - Strings
 - special treatment given to 190
 - StringTokenizer
 - use of 635
 - stubbing 12
 - super 129
 - super method
 - passing arguments to base class
 - constructor via 273
 - use of 270
 - supertypes & subtypes
 - reasoning about 723
 - Swing 305, 350
 - separable model architecture
 - 379, 389
 - Swing component
 - when to Paint() 384
 - Swing component painting
 - explanation of events 385
 - switch 129
 - switch statement 159
 - synchronized 129
 - synchronized code block 462
 - synchronized methods 466
- T**
- tagging interface
 - concept of 351
 - TCP 565
 - TCP/IP 554, 555, 556, 564–567
 - application layer 564
 - data link layer 565
 - Java support for 565
 - network layer 565
 - physical layer 565
 - transport layer 565
 - Ternary 145
 - test driver program 223
 - testing
 - user-defined type 223
 - TextPad™ 30
 - The 639
 - the art of programming 4, 8
 - inspiration 9
 - money but no time 9
 - mood setting 10
 - time but no money 9
 - where not to start 9
 - your computer 9
 - thinking outside the box 210
 - this 129
 - thread 443
 - analogy 444
 - blocked 456
 - Clock1 class 446
 - computing Pi 452
 - consumer 468
 - deadlock 474
 - definition of 444
 - interrupting 448
 - lock 460
 - main 444
 - priority 455
 - producer 468
 - race condition 460
 - race conditions 459
 - scheduling 455
 - sleeping 448
 - synchronization 460
 - synchronization rules 462–466
 - synchronized keyword 462
 - synchronizing methods 466
 - Thread Class 581
 - Thread class
 - constructor methods 449
 - methods of 447
 - notify() methods 470
 - run() method 449
 - start() method 449
 - wait() methods 470
 - yield() method 454
 - ThreadGroups 444
 - threads 444–477
 - creating your own 449
 - of a simple Java program 445
 - throw 129
 - throw keyword 436, 437
 - Throwable
 - manipulating object 433
 - methods of 433
 - Throwable class
 - catching all errors with 432
 - Throwable class hierarchy 428
 - throwing exceptions 436
 - throws 129
 - throws clause 436
 - tight spiral development 61
 - timeless way 746
 - Toolkit class 595
 - top-level container
 - API 312
 - constructors 312
 - methods 314
 - top-level containers 308
 - toString() method
 - testing 705
 - transient 129
 - transitivity
 - exhibited by inheritance hierar-
chies 267
 - Transmission Control Protocol
(TCP) 565
 - transparency
 - using to render GUI components
413
 - TreePrinterUtils class
 - use of 317
 - true 129
 - try 129
 - try/catch block 431
 - try/finally block
 - good use of 436
 - type 267, 297
 - type coercion 274
 - types
 - array 182
 - typesafe enumeration
 - example code 254
 - when to use 254
 - typesafe enumeration pattern 254
- U**
- UDP 554, 565
 - UML 13, 210, 244, 260
 - class diagram 213
 - composite aggregation 247, 249
 - expressing abstract class 277
 - expressing inheritance 268
 - expressing interfaces 280
 - expressing realization 280
 - expression aggregation 246
 - realization
 - diagram
 - expanded form 281
 - simple form 281
 - sequence diagram
 - aircraft engine object creation

- 254
 - sequence diagrams 250, 251
 - simple aggregation 246, 247
 - using to tame conceptual complexity 244
 - UML class diagram
 - purpose of 213
 - UnicastRemoteObject 599
 - example use of 597
 - usage of 571
 - UNICODE 29
 - Unified Modeling Language (UML) 13
 - UNIX
 - killing processes 560
 - listing processes 560
 - URI 554
 - URL 554
 - URL class
 - use of 568
 - User Datagram Protocol (UDP) 565
 - user-defined types 211
 - UTF Strings 584
 - reading 584
 - writing 584
 - UTFDataFormatException 584
 - utility methods
 - definition of 216
- V**
- variable 65
 - declaring in main() method 131
 - definition of 131
 - final 132
 - usage in main() method example 131
 - Vector
 - serializing to disk 519
 - verb phrases 64
 - verbs 64
 - vertical access 283
 - view 751
 - virtual machine 102
 - Java classic vs. HotSpot 103
 - Java HotSpot 102
 - void 129
 - volatile 129
- W**
- while 129
 - whole object 245
 - whole objects 246
 - whole/part class relationship 245
 - window 309
 - close options 311
 - decorations 310
 - word 96, 97
 - word processor
 - using to create source files 29
 - world
 - imperfect understanding of 680
 - wrapper classes 149
 - usage of 134
 - Writer class 510
 - Writers 508
- X**
- x
 - screen axis 308
 - XML
 - property files 529
- Y**
- y
 - screen axis 308

MASTER THE COMPLEXITIES OF JAVA AND OBJECT-ORIENTED PROGRAMMING

JAVA FOR ARTISTS: THE ART, PHILOSOPHY, AND SCIENCE OF OBJECT-ORIENTED PROGRAMMING TOWERS HEAD AND SHOULDERS ABOVE ORDINARY INTRODUCTORY JAVA TEXTS IN ITS COMPREHENSIVE COVERAGE AND AUDACIOUS STYLE. **JAVA FOR ARTISTS** IS THE ONLY BOOK OF ITS KIND THAT SUCCINCTLY ADDRESSES THE ART, PHILOSOPHY, AND SCIENCE OF JAVA AND OBJECT-ORIENTED PROGRAMMING. THE AUTHORS, BOTH CERTIFIED JAVA DEVELOPERS, ARE UNIQUELY QUALIFIED TO PRESENT MATERIAL OTHER BOOKS SHY AWAY FROM OR IGNORE COMPLETELY. **JAVA FOR ARTISTS** WILL HELP YOU SMASH THROUGH THE BARRIERS PREVENTING YOU FROM MASTERING THE COMPLEXITIES OF OBJECT-ORIENTED PROGRAMMING IN JAVA. START TAMING COMPLEXITY NOW! READ **JAVA FOR ARTISTS** TODAY – ENERGIZE YOUR PROGRAMMING SKILLS FOR LIFE!

SUPERCHARGE YOUR CREATIVE ENERGY BY RECOGNIZING AND UTILIZING THE POWER OF THE “FLOW”
LEARN A DEVELOPMENT CYCLE YOU CAN ACTUALLY USE AT WORK
COMPREHENSIVE PROGRAMMING PROJECT WALK-THROUGH SHOWS YOU HOW TO APPLY THE DEVELOPMENT CYCLE PROJECT APPROACH STRATEGY HELPS YOU MAINTAIN PROGRAMMING PROJECT MOMENTUM
JAVA STUDENT SURVIVAL GUIDE HELPS YOU TACKLE ANY PROJECT THROWN AT YOU
APPLY REAL-WORLD PROGRAMMING TECHNIQUES TO PRODUCE PROFESSIONAL CODE
IN-DEPTH COVERAGE OF ARRAYS ELIMINATES THEIR MYSTERY
CREATE COMPLEX GUIs USING SWING COMPONENTS
LEARN THE SECRETS OF THREAD PROGRAMMING TO CREATE MULTI-THREADED APPLICATIONS
MASTER THE COMPLEXITIES OF JAVA GENERICS AND LEARN HOW TO CREATE GENERIC METHODS
DISCOVER THREE OBJECT-ORIENTED DESIGN PRINCIPLES THAT WILL GREATLY IMPROVE YOUR SOFTWARE ARCHITECTURES
LEARN HOW TO DESIGN WITH INHERITANCE AND COMPOSITION TO CREATE FLEXIBLE AND RELIABLE SOFTWARE
CREATE WELL-BEHAVED JAVA OBJECTS THAT CAN BE USED PREDICTABLY AND RELIABLY IN JAVA APPLICATIONS
MASTER THE USE OF THE SINGLETON, FACTORY, MODEL-VIEW-CONTROLLER, AND COMMAND SOFTWARE DESIGN PATTERNS
REINFORCE YOUR LEARNING WITH THE HELP OF CHAPTER LEARNING OBJECTIVES, SKILL-BUILDING EXERCISES, SUGGESTED PROJECTS, AND SELF-TEST QUESTIONS
PACKED WITH 134 TABLES, 477 FIGURES, AND 410 CODE EXAMPLES CONTAINING OVER 7500 LINES OF CODE
ALL CODE EXAMPLES WERE COMPILED, EXECUTED, AND TESTED BEFORE BEING USED IN THE BOOK TO ENSURE QUALITY
AND MUCH, MUCH, MORE...!

“This book is SWEET!”

“JAVA FOR ARTISTS RAISES THE BAR FOR INTRODUCTORY JAVA TEXTS.”

“TIRED OF THE DEPRESSING SAMENESS OF ORDINARY JAVA BOOKS? GET JAVA FOR ARTISTS – IT DELIVERS THE GOODS!”

Rick Miller is a SENIOR COMPUTER SCIENTIST AND Web Applications Architect for Science Applications International Corporation (SAIC), and an ASSISTANT PROFESSOR at Northern Virginia Community College, Annandale Campus, where he teaches a variety of computer programming courses. He’s a certified JAVA programmer and developer.

Raffi Kasparian is a SOFTWARE ENGINEER AND Web Applications Developer for Science Applications International Corporation (SAIC). Raffi holds a DOCTOR OF MUSICAL ARTS (DMA) degree from the University of Michigan and plays the piano for the UNITED STATES ARMY MEN’S CHORUS. He’s a certified JAVA programmer and developer.

**“READ JAVA FOR ARTISTS Today –
ENERGIZE YOUR PROGRAMMING skills
FOR LIFE!”**



Pulp FREE PRESS

ISBN-13: 978-1-9325-0405-7

ISBN 1-932504-05-2



9 781932 504057

\$79.95