C# For Artists

The Art, Philosophy, And Science Of Object-Oriented Programming

Value Objects (VOs)

Business Layer (BL)

Data Access Layer (DAL)

Remote Object

BOs

DAOs

Microsoft SQL Server

.NET Remoting Infrastructure

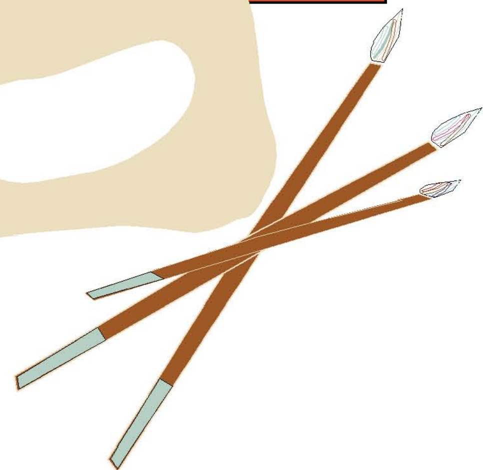
ADO.NET

Microsoft Enterprise Library

Employee Training Server Application

Covers Microsoft .NET 3.5

inheritance
patterns
threads
classes
arrays
interfaces
forms
boxing
polymorphism
database access
aggregation
expressions
collections
networking
statements



Rick Miller

Unleash The Creative Genius In You!

C# For Artists

C# For Artists

The Art, Philosophy, and Science of Object-Oriented Programming

Rick Miller

Pulp Free Press
Falls Church, Virginia

Pulp Free Press, Falls Church, Virginia 22042
www.pulpfreepress.com
info@pulpfreepress.com
©2008 Richard Warren Miller & 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 2008 - Regenerated 2010 with corrections
16 14 12 10 10 9 8 7 6 5 4 3

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 -
C# For Artists: The Art, Philosophy, and Science of Object-Oriented
Programming/Rick Miller
p. cm.
Includes bibliographical references and index.
ISBN-13: 9781932504071 ISBN-10: 1-932504-07-9 (pbk)
1. C#(computer program language)  II. Title.
2. Object-Oriented Programming (Computer Science)
QA76.73.C154 M555 2008
005.13’3--dc21         Library of Congress Control Number: 2006901642
                                                                  CIP
```


The source code provided in this book is intended for instructional purposes only. Although every possible measure was made by the author 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.

C# For Artists was meticulously crafted on a Macintosh PowerMac G4 and G5 using Adobe FrameMaker, Adobe Illustrator, Macromedia Freehand, Adobe Photoshop, Adobe Acrobat, Microsoft Word, and Maple. C# source code examples were prepared using TextPad, NotePad++, and Microsoft Visual Studio. Photographs appearing at the beginning of each chapter were made with a variety of cameras and film as noted in the vertical captions.

Printed in the United States of America.

ISBN-10: 1-932504-07-9; ISBN-13: 9781932504071 — First Paperback Edition

To all friends who make life agreeable.

Rick

Detailed Contents

Preface

Welcome – And Thank You! xliii
Target Audience xliii
Approach(es) xliii
Pedagogy – I Mean, How This Book's Arranged xliv
Learning Objectives xliv
Introduction xliv
Content xliv
Quick Reviews xliv
Summary xliv
Skill-Building Exercises xliv
Suggested Projects xlv
Self-Test Questions xlv
References xlv
Notes xlv
Typographical Formats xlv
This Is An Example Of A First Level Subheading xlv
This Is An Example Of A Second Level Subheading xlv
Source Code Formatting xlv
SupportSite™ Website xlvi
Problem Reporting xlvi
About The Author xlvi
Acknowledgments xlvi

1 An Approach To The Art Of Programming

Introduction 4
The Difficulties You Will Encounter Learning C# 4
Required Skills 4
The Planets Will Come Into Alignment 4
How This Chapter Will Help You 5
Personality Traits Found In Great Programmers 5
Creative 5
Tenacious 5
Resilient 5
Methodical 5
Meticulous 6
Honest 6
Proactive 6
Humble 6
Be a Generalist and a Just-in-Time Specialist 6

© 2008 Rick Miller — All Rights Reserved

Project Management **6**
Three Software Development Roles 6
Analyst *6*
Architect *7*
Programmer *7*
A Project-Approach Strategy 7
You Have Been Handed A Project – Now What? *7*
Strategy Areas of Concern *8*
Think Abstractly *9*
The Strategy In A Nutshell 10
Applicability To The Real World 10
The Art Of Programming **10**
Don't Start At The Computer 10
Inspiration Strikes At The Weirdest Time 10
Own Your Own Computer 11
You Either Have Time and No Money, or Money and No Time *11*
The Family Computer Is Not Going To Cut It! *11*
Set The Mood 11
Location, Location, Location *11*
Concept Of The Flow 11
The Stages of Flow *12*
Be Extreme 12
The Programming Cycle *12*
The Programming Cycle Summarized *13*
A Helpful Trick: Stubbing 13
Fix The First Compiler Error First 14
Managing Project Complexity **14**
Conceptual Complexity 14
Managing Conceptual Complexity *14*
The Unified Modeling Language (UML) *15*
Physical Complexity 15
Managing Physical Complexity *15*
The Relationship Between Physical and Conceptual Complexity 15
Maximize Cohesion – Minimize Coupling 15
Summary **16**
Skill-Building Exercises **16**
Suggested Projects **16**
Self-Test Questions **17**
References **17**
Notes **17**

2 Small Victories: Creating C# Projects

Introduction **20**
Creating Projects With Microsoft C#.NET Command-Line Tools **20**
Downloading And Installing The .NET Framework 20
Downloading And Installing Notepad++ 22
Configuring Your Development Environment 22
Environment Variables *22*
Creating A Project Folder *25*
Setting Folder Options *25*
Creating A Shortcut To The Command Console And Setting Its Properties *26*
Testing The Configuration 29
Creating The Source File *29*
Compiling The Source File *29*

© 2008 Rick Miller — All Rights Reserved

Executing The Application 30
Quick Review 31
Creating Projects With Microsoft Visual C# Express 32
Download and Install Visual C# Express 32
Quick Tour Of Visual C# Express 33
Select Project Type 33
Saving The Project 36
Build The Project 36
Locating The Project Executable File 36
Execute The Project 38
Where To Go For More Information About Visual C# Express 38
Quick Review 38
Summary 38
Skill-Building Exercises 39
Suggested Projects 39
Self-Test Questions 39
References 40
Notes 40

3 Project Walkthrough

Introduction 42
The Project-Approach Strategy Summarized 42
Development Cycle 43
Project Specification 44
Analyzing The Project Specification 45
Application Requirements Strategy Area 45
Problem-Domain Strategy Area 46
Language-Features Strategy Area 48
Design Strategy Area 50
Development Cycle: First Iteration 51
Plan (First Iteration) 51
Code (First Iteration) 52
Test (First Iteration) 52
Integrate/Test (First Iteration) 52
Development Cycle: Second Iteration 52
Plan (Second Iteration) 53
Code (Second Iteration) 53
Test (Second Iteration) 53
Integrate/Test (Second Iteration) 54
Development Cycle: Third Iteration 54
Plan (Third Iteration) 54
Code (Third Iteration) 55
Integrate/Test (Third Iteration) 57
A Bug In The Program 57
Development Cycle: Fourth Iteration 59
Plan (Fourth Iteration) 59
Implementing State Transition Diagrams 60
Implementing The PrintFloor() Method 60
Code (Fourth Iteration) 61
Test (Fourth Iteration) 62
Integrate/Test (Fourth Iteration) 63
Development Cycle: Fifth Iteration 63
Plan (Fifth Iteration) 63

© 2008 Rick Miller — All Rights Reserved

Code (Fifth Iteration) 64
Test (Fifth Iteration) 65
Integrate/Test (Fifth Iteration) 65
Final Considerations 66
Complete RobotRat.cs Source Code Listing 67
Summary 73
Skill-Building Exercises 73
Suggested Projects 73
Self-Test Questions 73
References 74
Notes 74

4 Computers, Programs, And Algorithms

Introduction 76
What Is A Computer? 76
Computer vs. Computer System 76
Computer System.................... 76
Processor 78
Three Aspects of Processor Architecture 79
Feature Set.................... 79
Feature Set Implementation.................... 79
Feature Set Accessibility 79
Memory Organization 79
Memory Basics 80
Memory Hierarchy 80
Bits, Bytes, Words 80
Alignment and Addressability 81
What Is A Program? 82
Two Views of a Program 82
The Human Perspective 82
The Computer Perspective.................... 82
The Processing Cycle 82
Fetch 83
Decode 83
Execute 83
Store 83
Why A Program Crashes 83
Algorithms 83
Good vs. Bad Algorithms 83
Don't Reinvent The Wheel! 86
Virtual Machines And The Common Language Infrastructure 86
Virtual Machines 87
The Common Language Infrastructure (CLI) 87
Four Parts Of The Common Language Infrastructure 87
The Cross Platform Promise 89
Summary 90
Skill-Building Exercises 90
Suggested Projects 91
Self-Test Questions 91
References 92
Notes 92

 © 2008 Rick Miller — All Rights Reserved

5. Navigating .NET Framework Documentation

Introduction **94**
MSDN: The Definitive Source For API Information **94**
Discovering Information About Classes **96**
General Overview Page 96
Class Member Page 97
Getting Information On Other Class Members 98
Quick Review 100
The Base Class Libraries (BCL) **100**
Quick Review 101
Navigating An Inheritance Hierarchy **101**
Quick Review 102
Beware Obsolete APIs **102**
Summary **103**
Skill-Building Exercises **103**
Suggested Projects **104**
Self-Test Questions **104**
References **104**
Notes **105**

6 Simple C# Programs

Introduction **110**
What Is A C# Program? **110**
A Simple Console Application **111**
Definition Of Terms: Application, Assembly, Module, and Entry Point 111
Structure Of A Simple Application 111
Purpose Of The Main() Method 112
Main() Method Signatures 112
Quick Review 113
Identifiers And Reserved Keywords **113**
Identifier Naming Rules 114
Quick Review 115
Types **115**
Value Type Variables vs. Reference Type Variables 116
Value Type Variables *116*
Reference Type Variables *116*
Maybe Some Pictures Will Help *117*
Mapping Predefined Types To System Structures *118*
Quick Review 119
Statements, Expressions, and Operators **119**
Statement Types 119
Operators And Their Use 120
Operator Precedence and Associativity *121*
Forcing Operator Precedence and Associativity Order With Parentheses *121*
Operators and Operands *121*
Operator Usage Examples 122
Primary Expression Operators *122*
Unary Expression Operators *122*
Multiplicative Expression Operators *123*
Additive Expression Operators *124*
Shift Expression Operators *124*
Relational, Type-Testing, and Equality Expression Operators *125*

 © 2008 Rick Miller — All Rights Reserved

Logical AND, OR, and XOR Expression Operators 126
Conditional AND and OR Expression Operators 129
Conditional (Ternary) Expression Operator 129
Assignment Expression Operators 130
Quick Review 131
Summary **131**
Skill-Building Exercises **131**
Suggested Projects **132**
Self-Test Questions **132**
References **133**
Notes **133**

7 Controlling The Flow Of Program Execution

Introduction **136**
Selection Statements **136**
If Statement 136
Handling Program Error Conditions 137
Executing Code Blocks In If Statements 139
Executing Consecutive If Statements 139
If/Else Statement 140
Chained If/Else Statements 141
Switch Statement 142
Implicit Case Fall-Through 143
Nested Switch Statement 144
Quick Review 145
Iteration Statements **145**
While Statement 145
Personality Of The While Statement 145
Do/While Statement 146
Personality Of The Do/While Statement 147
For Statement 148
How The For Statement Is Related To The While Statement 148
Personality Of The For Statement 148
Nesting Iteration Statements 149
Mixing Selection And Iteration Statements: A Powerful Combination 150
Quick Review 151
Break, Continue, And Goto **151**
Break Statement 152
Continue Statement 152
Goto Statement 153
Quick Review 153
Selection And Iteration Statement Selection Table **154**
Summary **155**
Skill-Building Exercises **155**
Suggested Projects **157**
Self-Test Questions **158**
References **159**
Notes **159**

8 Arrays

Introduction **162**
What Is An Array? **162**

© 2008 Rick Miller — All Rights Reserved

Specifying Array Types 163
Quick Review 164
Functionality Provided By C# Array Types 164
Array-Type Inheritance Hierarchy 164
Special Properties Of C# Arrays 165
Quick Review 165
Creating And Using Single-Dimensional Arrays 166
Arrays Of Value Types 166
How Value-Type Array Objects Are Arranged In Memory 166
Finding An Array's Type, Rank, And Total Number of Elements 167
Creating Single-Dimensional Arrays Using Array Literal Values 168
Differences Between Arrays Of Value Types And Arrays Of Reference Types 169
Single-dimensional Arrays In Action 171
Message Array 171
Calculating Averages 173
Histogram: Letter Frequency Counter 173
Quick Review 175
Creating And Using Multidimensional Arrays 176
Rectangular Arrays 176
Initializing Rectangular Arrays With Array Literals 178
Ragged Arrays 178
Multidimensional Arrays In Action 179
Weighted Grade Tool 179
Quick Review 181
The Main() Method's String Array 181
Purpose And Use Of The Main() Method's String Array 181
Manipulating Arrays With The System.Array Class 182
Numeric Formatting 183
Summary 183
Skill-Building Exercises 184
Suggested Projects 184
Self-Test Questions 187
References 188
Notes 188

9 Toward Problem Abstraction: Creating New Data Types

Introduction 190
Abstraction: Amplify The Essential, Eliminate The Irrelevant 190
Abstraction Is The Art Of Programming 190
Where Problem Abstraction Fits Into The Development Cycle 191
Creating Your Own Data Types 191
Case-Study Project: Write A People Manager Program 191
Quick Review 193
The UML Class Diagram 193
Quick Review 194
Overview Of The Class Construct 194
Eleven Categories Of Class Members 194
Fields 195
Constants 197
The Difference Between `const` and `readonly`; Compile-Time vs. Runtime Constants 197
Properties 198
Methods 199
Instance Constructors 199

© 2008 Rick Miller — All Rights Reserved

- Static Constructors 200
- Events 200
- Operators 200
- Indexers 200
- Nested Type Declarations 200
- Finalizers 200
- Access Modifiers 201
- Public 201
- Private 201
- Protected 201
- Internal 201
- Protected Internal 201
- The Concepts Of Horizontal Access, Interface, and Encapsulation 201
- Quick Review 202

Methods 202
- Method Naming: Use Action Words That Indicate The Method's Purpose 203
- Maximize Method Cohesion 203
- Structure Of A Method Definition 203
- Method Modifiers (optional) 203
- Return Type Or Void (optional) 204
- Method Name (mandatory) 205
- Parameter List (optional) 205
- Method Body (optional for abstract or external methods) 205
- Method Definition Examples 205
- Method Signatures 206
- Overloading Methods 206
- Constructor Methods 206
- Quick Review 206

Building And Testing The Person Class 207
- Start By Creating The Source File And Class Definition Shell 207
- Defining Person Instance Fields 207
- Defining Person Properties And Constructor Method 208
- Adding Properties 208
- Adding A Constructor Method 208
- Testing The Person Class: A Miniature Test Plan 209
- Use The PeopleManagerApplication Class As A Test Driver 209
- Adding Features To The Person Class: Calculating Age 210
- Adding Features To The Person Class: Convenience Properties 211
- Adding Features To The Person Class: Finishing Touches 213
- Quick Review 214

Building and Testing The PeopleManager Class 215
- Defining The PeopleManager Class Shell 215
- Defining PeopleManager Fields 215
- Defining PeopleManager Constructor Methods 215
- Defining Additional PeopleManager Methods 216
- Testing The PeopleManager Class 217
- Adding Features To The PeopleManager Class 217
- Quick Review 219

More About Methods 219
- Value Parameters And Reference Parameters 219
- Value Parameters: The Default Parameter Passing Mode 219
- Reference Parameters: Using The `ref` Parameter Modifier 220
- The `out` Parameter Modifier 223
- Parameter Arrays: Using The `params` Modifier 223
- Local Variable Scoping 224

© 2008 Rick Miller — All Rights Reserved

Anywhere An Object Of <type> Is Required, A Method That Returns <type> Can Be Used 224
Quick Review 225
Structures vs. Classes 225
Value Semantics vs. Reference Semantics 225
Ten Authorized Members vs. Eleven 226
Default Variable Field Values 226
Behavior During Assignment 226
`this` Behaves Differently 226
Inheritance Not Allowed 226
Boxing And Unboxing 226
When To Use Structures 227
Summary 227
Skill-Building Exercises 228
Suggested Projects 229
Self-Test Questions 231
References 232
Notes 232

10 Compositional Design

Introduction 234
Managing Conceptual And Physical Complexity 234
Compiling Multiple Source Files Simultaneously With csc 234
Quick Review 235
Dependency vs. Association 235
Aggregation 235
Simple vs. Composite Aggregation 236
The Relationship Between Aggregation And Object Lifetime 236
Quick Review 236
Expressing Aggregation In A UML Class Diagram 236
Simple Aggregation Expressed In UML 237
Composite Aggregation Expressed In UML 237
Aggregation Example Code 237
Simple Aggregation Example 238
Composite Aggregation Example 239
Quick Review 240
Sequence Diagrams 240
Magic Draw 241
Quick Review 241
The Engine Simulation: An Extended Example 242
The Purpose Of The Engine Class 243
Engine Class Attributes And Methods 244
Engine Simulation Sequence Diagrams 244
Running The Engine Simulation Program 244
Quick Review 246
Complete Engine Simulation Code Listing 246
Summary 250
Skill-Building Exercises 250
Suggested Projects 252
Self-Test Questions 252
References 253
Notes 253

© 2008 Rick Miller — All Rights Reserved

11 Inheritance and Interfaces

Introduction **256**
Three Purposes Of Inheritance **256**
Implementing The "is a" Relationship 257
The Relationship Between The Terms Type, Interface, and Class 257
Meaning Of The Term Interface *257*
Meaning Of The Term Class *257*
Quick Review 258
Expressing Generalization And Specialization In The UML **258**
A Simple Inheritance Example **259**
The UML Diagram 259
BaseClass Source Code 259
DerivedClass Source Code 260
DriverApplication Program 260
Quick Review 261
Another Inheritance Example: Person - Student **261**
The Person - Student UML Class Diagram 261
Person - Student Source Code 262
Casting 264
Use Casting Sparingly *265*
Quick Review 265
Overriding Base Class Methods **266**
Quick Review 267
Abstract Methods and Abstract Base Classes **267**
The Primary Purpose Of An Abstract Base Class 268
Expressing Abstract Base Classes In UML 268
Quick Review 270
Interfaces **270**
The Purpose Of Interfaces 270
Authorized Interface Members 270
The Differences Between An Interface And An Abstract Class 271
Expressing Interfaces In UML 271
Expressing Realization In A UML Class Diagram 271
An Interface Example 272
Quick Review 274
Controlling Horizontal And Vertical Access **274**
Quick Review 274
Sealed Classes And Methods **274**
Quick Review 274
Polymorphic Behavior **275**
Quick Review 275
Inheritance Example: Employee **275**
Inheritance Example: Engine Simulation **278**
Engine Simulation UML Diagram 278
Simulation Operational Description 278
Compiling The Engine Simulation Code 280
Complete Engine Simulation Code Listing **280**
Summary **284**
Skill-Building Exercises **285**
Suggested Projects **285**
Self-Test Questions **287**
References **287**
Notes **288**

© 2008 Rick Miller — All Rights Reserved

12 Windows Forms Programming

Introduction **292**
The Form Class **292**
- Form Class Inheritance Hierarchy 292
- A Simple Form Program 293
- Quick Review 294

Application Messages, Message Pump, Events, And Event Loop **294**
- Message Categories 295
- Messages In Action: Trapping Messages With IMessageFilter 296
- Final Thoughts On Messages 296
- Quick Review 297

Screen And Window (Client) Coordinate System **297**
- Quick Review 299

Manipulating Form Properties **299**
- Quick Review 301

Adding Components To Windows: Button, TextBox, And Label **301**
- Quick Review 302

Registering Event Handlers With GUI Components **303**
- Delegates And Events 303
- Quick Review 305

Handling GUI Component Events In Separate Objects **305**
- Quick Review 307

Layout Managers **307**
- FlowLayoutPanel 308
- TableLayoutPanel 310
- Quick Review 311

Menus **312**
- Quick Review 315

A Little More About TextBoxes **315**
- Quick Review 316

The Rhythm Of Coding GUIs **317**
Summary **317**
Skill-Building Exercises **318**
Suggested Projects **319**
Self-Test Questions **319**
References **320**
Notes **320**

13 Custom Events

Introduction **322**
C# Event Processing Model: An Overview **322**
- Quick Review 323

Custom Events Example: Minute Tick **323**
Custom Events Example: Automated Water Tank System **326**
Naming Conventions **331**
Final Thoughts On Extending The EventArgs Class **332**
Summary **332**
Skill-Building Exercises **333**
Suggested Projects **333**
Self-Test Questions **333**
References **334**

© 2008 Rick Miller — All Rights Reserved

Notes ... 334

14 Collections

Introduction ... 338
Case Study: Building A Dynamic Array ... 338
Evaluating DynamicArray ... 340
The ArrayList Class To The Rescue ... 340
A Quick Peek At Generics ... 341
Quick Review ... 341
Data Structure Performance Characteristics ... 342
Array Performance Characteristics ... 342
Linked List Performance Characteristics ... 343
Hash Table Performance Characteristics ... 345
Chained Hash Table vs. Open-Address Hash Table ... 345
Red-Black Tree Performance Characteristics ... 346
Stacks And Queues ... 347
Quick Review ... 347
Navigating The .NET Collections API ... 348
System.Collections ... 348
System.Collections.Generic ... 348
System.Collections.ObjectModel ... 349
System.Collections.Specialized ... 349
Mapping Non-Generic To Generic Collections ... 349
Quick Review ... 350
Using Non-Generic Collection Classes - Pre .NET 2.0 ... 350
Objects In – Objects Out: Casting 101 ... 351
Extending ArrayList To Create A Strongly-Typed Collection ... 352
Using Generic Collection Classes – .NET 2.0 and Beyond ... 354
List<T>: Look Ma, No More Casting! ... 354
Implementing KeyedCollection<TKey, TItem> ... 355
Quick Review ... 356
Special Operations On Collections ... 357
Sorting A List ... 357
Implementing System.IComparable<T> ... 357
Extending Comparer<T> ... 359
Converting A Collection Into An Array ... 361
Quick Review ... 361
Summary ... 362
Skill-Building Exercises ... 362
Suggested Projects ... 363
Self-Test Questions ... 363
References ... 364
Notes ... 364

15 Exceptions: Writing Fault-Tolerant Software

Introduction ... 366
What Is An Exception ... 366
.NET CLR Exception Handling Mechanism ... 366
Unhandled Exceptions ... 366
The Exception Information Table ... 367
Quick Review ... 367

© 2008 Rick Miller — All Rights Reserved

Exception Class Hierarchy **367**
Application vs. Runtime Exceptions 368
Runtime Exception Listing 368
Determining What Exceptions A .NET Framework Method Throws 369
Quick Review 369
Exception Class Properties **370**
Quick Review 370
Creating Exception Handlers: Using Try/Catch/Finally Blocks **371**
Using A Try/Catch Block 371
First Line of Defense: Use Defensive Coding 372
Using Multiple Catch Blocks 372
Using A Finally Block 373
Quick Review 374
Creating Custom Exceptions **374**
Extending The Exception Class 374
Manually Throwing An Exception With The throw Keyword 375
Translating Low-Level Exceptions Into High-Level Exceptions 375
Quick Review 376
Documenting Exceptions **376**
Summary **377**
Skill-Building Exercises **377**
Suggested Projects **378**
Self-Test Questions **378**
References **378**
Notes **379**

16 Multithreaded Programming

Introduction **382**
Multithreading Overview: The Tale Of Two Vacations **382**
Single-Threaded Vacation 382
Multithreaded Vacation 382
The Relationship Between A Process And Its Threads 383
Vacation Gone Bad 384
Quick Review 385
Creating Managed Threads With The Thread Class **385**
Single-Threaded Vacation Example 386
Multithreaded Vacation Example 386
Thread States 389
Creating And Starting Managed Threads 389
ThreadStart Delegate 389
ParameterizedThreadStart Delegate: Passing Arguments To Threads 390
Blocking A Thread With Thread.Sleep() 391
Blocking A Thread With Thread.Join() 392
Foreground vs. Background Threads 394
Quick Review 395
Creating Threads With The BackgroundWorker Class **396**
Quick Review 399
Thread Pools **399**
Quick Review 400
Asynchronous Method Calls **400**
Obtaining Results From An Asynchronous Method Call 402
Providing A CallBack Method To BeginInvoke() 402

© 2008 Rick Miller — All Rights Reserved

Quick Review ... 403
Summary ... **404**
Skill-Building Exercises ... **405**
Suggested Projects ... **405**
Self-Test Questions ... **406**
References ... **406**
Notes ... **407**

17 File I/O

Introduction ... **410**
Manipulating Directories And Files ... **410**
Files, Directories, And Paths ... 411
Manipulating Directories And Files ... 411
Verbatim String Literals ... 412
Quick Review ... 413
Serializing Objects To Disk ... **413**
Serializable Attribute ... 413
Serializing Objects With BinaryFormatter ... 414
Serializing Objects With XMLSerializer ... 416
Quick Review ... 418
Working With Text Files ... **418**
Some Issues You Must Consider ... 418
Saving Dog Data To A Text File ... 418
Quick Review ... 420
Working With Binary Data ... **420**
Quick Review ... 422
Random Access File I/O ... **422**
Towards An Approach To The Adapter Project ... 422
Start Small And Take Baby Steps ... 423
Other Project Considerations ... 424
Locking A Record For Updates And Deletes ... 424
Monitor.Enter()/Monitor.Exit() vs. The lock Keyword ... 425
Translating Low-Level Exceptions Into Higher-Level Exception Abstractions ... 425
Where To Go From Here ... 425
Complete RandomAccessFile Legacy Datafile Adapter Source Code Listing ... 425
Quick Review ... 437
Working With Log Files ... **438**
Quick Review ... 440
Using FileDialogs ... **440**
Quick Review ... 442
Summary ... **443**
Skill-Building Exercises ... **444**
Suggested Projects ... **444**
Self-Test Questions ... **444**
References ... **445**
Notes ... **445**

18 Network Programming Fundamentals

Introduction ... **450**
What Is A Computer Network? ... **450**
Purpose Of A Network ... 450

© 2008 Rick Miller — All Rights Reserved

The Role Of Network Protocols 451
Homogeneous Vs. Heterogeneous Networks 451
The Unifying Network Protocols: TCP/IP 451
What's So Special About The Internet? 452
Quick Review 452
Servers & Clients 453
Server Hardware And Software 453
Client Hardware And Software 453
Quick Review 454
Application Distribution 454
Physical Distribution On One Computer 454
Running Multiple Clients On The Same Computer 454
Addressing The Local Machine 455
Physical Distribution Across Multiple Computers 455
Quick Review 455
Multitiered Applications 456
Logical Application Tiers 456
Physical Tier Distribution 456
Quick Review 456
Internet Networking Protocols: Nuts & Bolts 457
The Internet Protocols: TCP, UDP, And IP 457
The Application Layer 458
Transport layer 458
Network Layer 459
Data Link And Physical Layers 459
Putting It All Together 459
What You Need To Know 460
Quick Review 460
Summary 460
Skill-Building Exercises 461
Suggested Projects 462
Self-Test Questions 462
References 462
Notes 463

19 Networked Client -Server Applications

Introduction 466
Building Client-Server Applications With .NET Remoting 466
The Three Required Components of A .NET Remoting Application 466
A Simple .NET Remoting Application 467
SingleCall vs. Singleton 469
Accessing A Remote Object Via An Interface 470
Using Configuration Files 472
Passing Objects Between Client And Server 474
Quick Review 477
Client-Server Applications With TcpListener And TcpClient 478
TCP/IP Client-Server Overview 478
A Simple Client-Server Application 479
Building A Multithreaded Server 480
Listening On Multiple IP Addresses 482
Sending Objects Between Client And Server 484
Quick Review 488
Summary 489

© 2008 Rick Miller — All Rights Reserved

Skill-Building Exercises **490**
Suggested Projects **491**
Self-Test Questions **491**
References **492**
Notes **492**

20 Database Access & Multitiered Applications

Introduction **494**
What You Are Going To Build **494**
Preliminaries **495**
Installing SQL Server Express Edition 495
Installing Microsoft SQL Server Management Studio Express 496
Installing Microsoft Enterprise Library 498
A Simple Test Application 499
Introduction To Relational Databases And SQL **500**
Terminology 501
Structured Query Language (SQL) 502
Data Definition Language (DDL) 502
Creating The EmployeeTraining Database *502*
Creating A Database With A Script *503*
Creating Tables *504*
SQL Server Database Types *505*
Data Manipulation Language (DML) 506
Using The Insert Command *507*
Using The Select Command *507*
Using The Update Command *509*
Using The Delete Command *510*
Quick Review 511
Complex SQL Queries **511**
Creating A Related Table With A Foreign Key 511
Inserting Test Data Into The tbl_employee_training Table 512
Selecting Data From Multiple Tables 514
Join Operations *514*
Testing The Cascade Delete Constraint 515
Quick Review 515
The Server Application **516**
Project Folder Organization 516
Using Microsoft Build To Manage And Build the Project 517
First Iteration 519
Coding The EmployeeVO And EmployeeDAO *520*
Application Configuration File *528*
Creating Test Application *528*
Second Iteration 531
Testing The Code - Second Iteration *543*
Reality Check *551*
Third Iteration 551
The Client Application **556**
Third Iteration (continued) 556
Fourth Iteration 558
Fifth Iteration 564
Sixth Iteration 569
Compiling And Running The Modified EmployeeTrainingClient Project 580
Where To Go From Here 582

© 2008 Rick Miller — All Rights Reserved

Summary 583
Skill-Building Exercises 583
Suggested Projects 584
Self-Test Questions 584
References 585
Notes 585

21 Operator Overloading

Introduction 590
Operator Overloading 590
Overloadable Operators 590
Quick Review 591
Overloading Unary Operators 591
+,- Operators 591
! Operator 592
true, false Operators 593
++ --, Operators 595
Quick Review 597
Overloading Binary Operators 597
+, - Operators 597
*, / Operators 599
&, | Operators 601
Quick Review 603
Overloading Comparison Operators 603
==, !=,<,>,<=,>= Operators 604
Quick Review 607
Creating Implicit And Explicit Cast Operators 607
Implicit vs. Explicit Cast 607
Overloaded Cast Operators Example 607
Quick Review 610
The Assignment Operators: Things You Get For Free 610
Quick Review 610
Summary 610
Skill-Building Exercises 611
Suggested Projects 611
Self-Test Questions 611
References 612
Notes 612

22 Well-Behaved Objects

Introduction 614
Object Behavior Defined 614
Fundamental Behavior 614
Copy/Assignment Behavior 614
Equality Behavior 615
Comparison/Ordering Behavior 615
Seven Object Usage Scenarios 615
Fundamental Behavior 616
Object Creation – Constructors 616
Default Constructor 616
Private Constructors 616

© 2008 Rick Miller — All Rights Reserved

Overloaded Constructors 616
Member Accessibility 616
Horizontal Member Access 616
Vertical Member Access 617
Overriding Object.ToString() 617
Static vs. Instance Members 617
Serialization 618
Custom Serialization Example 618
Quick Review 623
Copy/Assignment Behavior 623
Value Object vs. Reference Object Assignment 624
Rule Of Thumb – Favor The Class Construct For Complex Types 624
Shallow Copy vs. Deep Copy 624
Copy Constructors 625
System.ICloneable vs. Object.MemberwiseClone() 627
Quick Review 629
Equality Behavior 629
Reference Equality vs. Value Equality 629
Rules For Overriding The Object.Equals() Method 630
Overriding The Object.GetHashCode() Method 630
Bloch's hash Code Generation Algorithm 631
Ashmore's Hash Code Generation Algorithm 631
Overridng Object.Equals() and Object.GetHashCode() Methods In The PersonVO Class 632
Quick Review 634
Comparison/Ordering Behavior 634
Implementing System.IComparable<T> 634
Rules For Implementing The CompareTo(T other) Method 635
Extending The Comparer<T> Class 636
Quick Review 638
Summary 638
Skill-Building Exercises 638
Suggested Projects 638
Self-Test Questions 639
References 639
Notes 640

23 Three Design Principles

Introduction 642
The Preferred Characteristics Of An Object-Oriented Architecture 642
Easy To Understand: How does this thing work? 642
Easy To Reason About: What are the effects of change? 642
Easy To Extend: Where do I add functionality? 642
The Liskov Substitution Principle & Design by Contract 643
Reasoning About The Behavior Of Supertypes And Subtypes 643
Relationship Between The LSP And DbC 643
The Common Goal Of The LSP And DbC 643
C# Support For The LSP And DbC 643
Designing With The LSP/DbC In Mind 644
Class Declarations Viewed As Behavior Specifications 644
Quick Review 644
Preconditions, Postconditions, And Class Invariants 644
Class Invariant 644
Precondition 644

© 2008 Rick Miller — All Rights Reserved

Postcondition 645
An Example 645
A Note On Using The Debug.Assert() Method To Enforce Pre- and Postconditions 646
Using Incrementer As A Base Class 646
Changing The Preconditions Of Derived Class Methods 648
Changing The Postconditions Of Derived Class Methods 652
Special Cases Of Preconditions And Postconditions 652
Method Argument Types 653
Method Return Types 654
Three Rules Of The Substitution Principle 654
Signature Rule 655
Methods Rule 655
Properties Rule 655
Quick Review 655
The Open-Closed Principle 656
Achieving The Open-Closed Principle 656
An OCP Example 656
Quick Review 661
The Dependency Inversion Principle 661
Characteristics Of Bad Software Architecture 661
Characteristics Of Good Software Architecture 662
Selecting The Right Abstractions Takes Experience 662
Quick Review 662
Terms and Definitions 663
Summary 663
Skill-Building Exercises 664
Suggested Projects 664
Self-Test Questions 664
References 665
Notes 666

24 Inheritance, Composition, Interfaces, Polymorphism

Introduction 668
Inheritance Vs. Composition: The Great Debate 668
What's The End Game? 669
Flexible Application Architectures 669
Modularity And Reliability 669
Architectural Stability Via Managed Dependencies 669
Knowing When To Accept A Design That's Good Enough 670
Quick Review 670
Inheritance-Based Design 670
Three Good Reasons To Use Inheritance 670
As A Means To Reason About Code Behavior 670
To Gain A Measure Of Code Reuse 670
To Facilitate Incremental Development 670
Forms Of Inheritance: Meyer's Inheritance Taxonomy 671
Coad's Inheritance Criteria 672
Person - Employee Example Revisited 673
Quick Review 673
The Role Of Interfaces 674
Reducing Or Limiting Intermodule Dependencies 674
Modeling Dominant, Collateral, and Dynamic Roles 674
Dominant Roles 674

© 2008 Rick Miller — All Rights Reserved

Collateral Roles *675*
Dynamic Roles *675*
Quick Review 675
Applied Polymorphism **675**
Quick Review 676
Composition-Based Design As A Force Multiplier **676**
Two Types Of Aggregation 676
Polymorphic Containment 676
An Extended Example **677**
Quick Review 682
Summary **682**
Skill-Building Exercises **683**
Suggested Projects **684**
Self-Test Questions **685**
References **685**
Notes **686**

25 Helpful Design Patterns

Introduction **688**
Software Design Patterns And How They Came To Be **688**
What Exactly Is A Software Design Pattern? 688
Origins 688
Pattern Specification 689
Applying Software Design Patterns 689
Quick Review 689
The Singleton Pattern **690**
Quick Review 693
The Factory Pattern **693**
The Dynamic Factory 693
Advantages Of The Dynamic Factory Pattern 695
Quick Review 695
The Model-View-Controller Pattern **695**
Quick Review 697
The Command Pattern **697**
Quick Review 702
A Comprehensive Pattern-Based Example **702**
Complete Code Listing 702
Com.PulpFreePress.Exceptions *702*
Com.PulpFreePress.Common *702*
Com.PulpFreePress.Utils *707*
Com.PulpFreePress.Commands *710*
Com.PulpFreePress.Model *713*
Com.PulpFreePress.View *714*
Com.PulpFreePress.Controller *720*
Running The Application 721
Summary **721**
Skill-Building Exercises **722**
Suggested Projects **723**
Self-Test Questions **723**
References **723**
Notes **724**

© 2008 Rick Miller — All Rights Reserved

Appendix A: Helpful Checklists And Tables

Project-Approach Strategy Check-off List 727
Development Cycle 728
Final Project Review Checklist 728

Appendix B: ASCII Table

ASCII Table 729

Appendix C: Identifier Naming: Writing Self-Commenting Code

Identifier Naming: Writing Self-Commenting Code 733
Benefits of Self-Commenting Code 733
Coding Convention 733
Class Names 733
Constant Names 734
Variable Names 734
Method Names 734
Property Names 735

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

List of Tables

Table 3-1: Project Approach Strategy 42
Table 3-2: Development Cycle 43
Table 3-3: Project Specification 44
Table 3-4: Robot Rat Nouns and Verbs 46
Table 3-5: Language Feature Study Check-Off List For Robot Rat Project 48
Table 3-6: First Iteration Design Considerations 51
Table 3-7: Second Iteration Design Considerations 53
Table 3-8: Third Iteration Design Considerations 54
Table 3-9: Fourth Iteration Design Considerations 59
Table 3-10: Fifth Iteration Design Considerations 63
Table 3-11: Final Project Review Checklist 66
Table 5-1: Base Class Library (BCL) Namespaces 100
Table 5-2: Additional .NET Libraries Used Heavily In This Book 100
Table 6-1: C# Reserved Keywords 113
Table 6-2: Predefined Type Mappings, Default Values, and Value Ranges 118
Table 6-3: C# Statement Types 120
Table 6-4: Operator Categories by Precedence 120
Table 6-5: Comparison Operator Behavior 125
Table 6-6: Logical Operator Behavior 126
Table 7-1: C# Selection And Iteration Statement Selection Guide 154
Table 8-1: C# Array Properties 165
Table 8-2: Numeric Formatting 183
Table 8-3: EISCS Machine Instructions 185
Table 9-1: People Manager Program Class Responsibilities 192
Table 9-2: Method Modifiers 203
Table 11-1: Differences Between Abstract Classes and Interfaces 271
Table 12-1: System Message Categories and their Prefixes 295
Table 12-2: Partial Listing of Control Events 303
Table 14-1: Mapping Non-Generic Collections to Their Generic Counterparts 349
Table 14-2: Rules For Implementing IComparable<T>.CompareTo(T other) Method 358
Table 15-1: Runtime Exceptions – Partial Listing 368
Table 15-2: Exception Class Public Properties 370
Table 20-1: SQL Server Data Types 505
Table 20-2: Project Folder Descriptions 516
Table 20-3: Employee Training Server Application – First Iteration Design Considerations & Decisions 519
Table 20-4: .NET to DbType to SQL Server Type to IDataReader Method Mapping 527
Table 20-5: Employee Training Server Application – Second Iteration Design Considerations And Decisions 532
Table 20-6: Employee Training Server Application – Third Iteration Design Considerations And Decisions 551
Table 20-7: Employee Training Client Application – Third Iteration Design Considerations And Decisions (Continued) 556
Table 20-8: Employee Training Client Application – Fourth Iteration Design Considerations And Decisions 559
Table 20-9: Employee Training Client Application – Fifth Iteration Design Considerations And Decisions 564
Table 20-10: Employee Training Client Application – Sixth Iteration Design Considerations And Decisions 569
Table 21-1: Overloadable Operators 590
Table 22-1: Object Usage Scenario Evaluation Checklist 615
Table 22-2: Rules for Overriding Object.Equals() method 630
Table 22-3: The GetHashCode() General Contract 630
Table 22-4: Rules For Implementing IComparable<T>.CompareTo(T other) Method 635
Table 23-1: Terms and Definitions Used in this Chapter 663
Table 24-1: Inheritance Form Descriptions 671
Table 25-1: Pattern Specification Template 689
Table 26-1: Project Approach Strategy 727
Table 26-2: Development Cycle 728

© 2008 Rick Miller — All Rights Reserved

Table 26-3: Final Project Review Checklist 728
Table Appendix B-1: ASCII Table 729
Table 26-1: Class Naming Examples 733
Table 26-2: Constant Naming Examples 734
Table 26-3: Variable Naming Examples 734
Table 26-4: Method Naming Examples 734
Table 26-5: Property Naming Examples 735

© 2008 Rick Miller — All Rights Reserved

List of Figures

Figure 1-1: Isomorphic Mapping Between Problem Domain and Design Domain 9
Figure 2-1: Microsoft.NET Framework Installation Directory 21
Figure 2-2: Partial Directory Listing of the v3.5 Folder 21
Figure 2-3: Creating an Environment Variable 23
Figure 2-4: Editing the Path User Environment Variable 24
Figure 2-5: Command Console Window 24
Figure 2-6: Testing the DOT_NET_FRAMEWORK_HOME Environment Variable 24
Figure 2-7: Testing the Path Environment Variable by Running the C# Compiler 25
Figure 2-8: Creating a New Folder 25
Figure 2-9: Projects Folder Before Setting Folder Options 26
Figure 2-10: Folder Options Dialog Window 26
Figure 2-11: Projects Folder After Setting Folder Options 27
Figure 2-12: Default Command Console Window 27
Figure 2-13: Command Console Properties Dialog 28
Figure 2-14: Setting the Start in Property 28
Figure 2-15: Setting Command Console Layout Properties 29
Figure 2-16: Directory Listing of the Chapter2 Directory Showing the HelloWorld.cs File 30
Figure 2-17: Compiling HelloWorld.cs Using the csc C# Compiler Command 30
Figure 2-18: Running the HelloWorld Program 30
Figure 2-19: Compiler Output Showing Compiler Error on Line 6 at Position 39 31
Figure 2-20: C# Language Compiler Errors 31
Figure 2-21: C# Compiler Error CS1002 "; expected" 32
Figure 2-22: Visual C# Express Installation Window 33
Figure 2-23: Visual C# Express Initial Start-Up Screen 34
Figure 2-24: New Project Dialog Showing Console Application Selected 34
Figure 2-25: HelloWorld Project View 35
Figure 2-26: IntelliSense Pop-Up Window Showing Available Console Object Methods and Properties 35
Figure 2-27: Updated HelloWorld Visual C# Project 36
Figure 2-28: Saving the HelloWorld Project 36
Figure 2-29: Building HelloWorld Project 37
Figure 2-30: Results of Running the tree /f Command from the Command Prompt 37
Figure 3-1: Tight-Spiral Development Cycle Deployment 43
Figure 3-2: Robot Rat Viewed as a Collection of Attributes 47
Figure 3-3: Robot Rat Floor Sketch 47
Figure 3-4: Complete Robot Rat Attributes 48
Figure 3-5: RobotRat UML Class Diagram 50
Figure 3-6: Compiling and Testing RobotRat – First Iteration 52
Figure 3-7: Compiling & Testing RobotRat - Second Iteration 54
Figure 3-8: Testing Menu Commands 57
Figure 3-9: A Disturbing Error Message 58
Figure 3-10: Unhandled IndexOutOfRangeException Error Message 58
Figure 3-11: pen_position State Transition Diagram 60
Figure 3-12: State Transition Diagram for the direction Variable 60
Figure 3-13: Testing the PrintFloor() Method 62
Figure 3-14: Testing Robot Rat Movement in All Directions 66
Figure 3-15: Robot Rat HTML Documentation Generated with Doxygen 72
Figure 4-1: Typical Apple Mac Pro Computer System 76
Figure 4-2: System Unit Components 77
Figure 4-3: Main Logic Board Block Diagram 77
Figure 4-4: Intel Xeon 5100 Dual core Processor 78
Figure 4-5: Intel Xeon 5100 Dual-Core Microprocessor Block Diagrams 78
Figure 4-6: Memory Hierarchy 80

© 2008 Rick Miller — All Rights Reserved

Figure 4-7: Simplified Memory Subsystem Diagram . . . 80
Figure 4-8: Simplified Main Memory Diagram . . . 81
Figure 4-9: Processing Cycle . . . 83
Figure 4-10: Dumb Sort Results 1 . . . 85
Figure 4-11: Dumb Sort Results 2 . . . 85
Figure 4-12: Dumb Sort Results 3 . . . 85
Figure 4-13: Algorithmic Growth Rates . . . 85
Figure 4-14: The C# Compile and Execution Process Overview . . . 86
Figure 4-15: MSIL Disassembler Session Showing Main() Method IL Instructions . . . 87
Figure 4-16: The Common Language Infrastructure Architecture . . . 88
Figure 4-17: Managed Assemblies can be Executed on any System that Implements the Common Language Infrastructure . . . 89
Figure 4-18: Chapter 3's Robot Rat Program Running in the Mono Environment on Apple OS X . . . 89
Figure 4-19: Microsoft .NET Architecture . . . 90
Figure 5-1: .NET Framework Class Library Reference Page . . . 94
Figure 5-2: .NET Development Link Expanded and Class Library Link Highlighted . . . 95
Figure 5-3: Class Library Link Expanded and System Namespace Highlighted . . . 95
Figure 5-4: String Class API Reference Overview Page . . . 96
Figure 5-5: String Members Page . . . 97
Figure 5-6: String Class's Public Constructors Partial Listing . . . 98
Figure 5-7: String Class's Methods Page Partial Listing . . . 98
Figure 5-8: String.SubString Method Page . . . 99
Figure 5-9: String.SubString Page with Collapsed Subheadings . . . 99
Figure 5-10: String.SubString Example Section Expanded Showing Example Code . . . 99
Figure 5-11: String Class Inheritance Hierarchy . . . 101
Figure 5-12: Obsolete .NET Framework Version 2.0 API Partial Listing by Namespace . . . 102
Figure 6-1: Results of Running Example 6.1 . . . 112
Figure 6-2: Results of Compiling Example 6.3 with Improper Main() Method Signature . . . 113
Figure 6-3: Errors Produced when Attempting to Reintroduce a Reserved Keyword . . . 114
Figure 6-4: C# Type Hierarchy . . . 115
Figure 6-5: Results of Running Example 6.6 . . . 116
Figure 6-6: The Results of Running Example 6.7 . . . 117
Figure 6-7: Value Type Memory Allocation . . . 117
Figure 6-8: Reference Type Memory Allocation . . . 117
Figure 6-9: Results of Calling the Append() Method via the sb1 Variable . . . 117
Figure 6-10: Results of Running Example 6.9 . . . 122
Figure 6-11: Results of Running Example 6.10 . . . 123
Figure 6-12: Results of Running Example 6.11 . . . 124
Figure 6-13: Results of Running Example 6.12 . . . 124
Figure 6-14: Results of Running Example 6.13 . . . 126
Figure 6-15: Logical AND, OR, and XOR Truth Tables . . . 127
Figure 6-16: Results of Running Example 6.14 . . . 127
Figure 6-17: Results of Running Example 6.15 . . . 128
Figure 6-18: Results of Running Example 6.16 . . . 129
Figure 6-19: Results of Running Example 6.17 . . . 129
Figure 6-20: Compiler Warning due to Unreachable Code . . . 130
Figure 6-21: Results of Running Example 6.18 . . . 130
Figure 6-22: Results of Running Example 6.19 . . . 131
Figure 7-1: if Statement Execution Diagram . . . 136
Figure 7-2: Results of Running Example 7.1 . . . 137
Figure 7-3: Typical .NET Error Message Dialog Window . . . 137
Figure 7-4: Unhandled IndexOutOfRangeException Message . . . 137
Figure 7-5: FormatException Error Message . . . 138
Figure 7-6: Results of Running Example 7.2 . . . 138
Figure 7-7: Results of Running Example 7.3 . . . 139
Figure 7-8: Results of Running Example 7.4 . . . 140
Figure 7-9: if/else Statement Execution Diagram . . . 140
Figure 7-10: Results of Running Example 7.5 . . . 141
Figure 7-11: Results of Running Example 7.6 . . . 142
Figure 7-12: switch Statement Execution Diagram . . . 142

© 2008 Rick Miller — All Rights Reserved

Figure 7-13: Results of Running Example 7.7 143
Figure 7-14: Results of Running Example 7.8 144
Figure 7-15: Results of Running Example 7.9 145
Figure 7-16: while Statement Execution Diagram 146
Figure 7-17: Results of Running Example 7.10 146
Figure 7-18: do/while Statement Execution Diagram 147
Figure 7-19: Results of Running Example 7-11 147
Figure 7-20: for Statement Execution Diagram 148
Figure 7-21: Results of Running Example 7.12 149
Figure 7-22: Results of Running Example 7.13 150
Figure 7-23: Results of Running CheckBookBalancer 151
Figure 7-24: Results of Running Example 7.15 152
Figure 7-25: Results of Running Example 7.16 153
Figure 7-26: Results of Running Example 7.17 153
Figure 8-1: Array Elements are Contiguous and Homogeneous 162
Figure 8-2: Declaring a Single-Dimensional Array 163
Figure 8-3: Array-Type Inheritance Hierarchy 164
Figure 8-4: Results of Running Example 8.1 166
Figure 8-5: Memory Representation of Value Type Array int_array Showing Default Initialization 167
Figure 8-6: Results of Running Example 8.2 167
Figure 8-7: Element Values of int_array After Initialization Performed by Second `for` Loop 168
Figure 8-8: Results of Running Example 8.3 168
Figure 8-9: Results of Running Example 8.4 169
Figure 8-10: Results of Running Example 8.5 170
Figure 8-11: State of Affairs After Line 5 of Example 8.5 Executes 170
Figure 8-12: State of Affairs After Line 10 of Example 8.5 Executes. 171
Figure 8-13: State of Affairs After Line 14 of Example 8.5 Executes 171
Figure 8-14: Final State of Affairs: All object_array Elements Point to an Object object 172
Figure 8-15: Results of Running Example 8.6 173
Figure 8-16: Results of Running Example 8.7 174
Figure 8-17: Results of Running Example 8.8 175
Figure 8-18: Rectangular Array Declaration Syntax 176
Figure 8-19: Accessing Two-Dimensional Array Elements 177
Figure 8-20: Results of Running Example 8.9 177
Figure 8-21: Results of Running Example 8.10 178
Figure 8-22: Array Declaration Syntax for a Two-Dimensional Ragged Array 179
Figure 8-23: Results of Running Example 8.11 179
Figure 8-24: Results of Running Example 8.12 181
Figure 8-25: Results of Running Example 8.13 182
Figure 8-26: Results of Running Example 8.14 183
Figure 9-1: People Management Program Project Specification 191
Figure 9-2: Class Diagram for People Manager Classes 193
Figure 9-3: Static and Non-Static Fields 195
Figure 9-4: Results of Running Example 9.1 196
Figure 9-5: Error Resulting from an Attempt to Assign to a Readonly Field 196
Figure 9-6: Results of Running Example 9.3 197
Figure 9-7: Results of Running Example 9,4 197
Figure 9-8: Results of Running Example 9.5 199
Figure 9-9: Horizontal Access Controlled via Access Modifiers public and private 202
Figure 9-10: Method Definition Structure 203
Figure 9-11: Results of Running Example 9.10 210
Figure 9-12: Results of Running Example 9.12 211
Figure 9-13: Results of Running Example 9.14 213
Figure 9-14: Results of Running Example 9.16 214
Figure 9-15: Results of Running Example 9.21 217
Figure 9-16: Results of Running Example 9.23 219
Figure 9-17: Default Value Parameter Behavior 220
Figure 9-18: Reference Parameter Behavior – Using `ref` Modifier 221
Figure 9-19: Results of Running Example 9.24 222

© 2008 Rick Miller — All Rights Reserved

Figure 9-20: Results of Running Example 9.25 . . . 222
Figure 9-21: Results of Running Example 9.26 . . . 223
Figure 9-22: Results of Running Example 9.27 . . . 224
Figure 9-23: Structures vs. Value Types . . . 225
Figure 9-24: Results of Running Example 9.28 . . . 227
Figure 9-25: Circular Linked List with Three Nodes . . . 230
Figure 10-1: UML Diagram Showing Simple Aggregation . . . 237
Figure 10-2: Part Class Shared Between Simple Aggregate Classes . . . 237
Figure 10-3: UML Diagram Showing Composite Aggregation . . . 237
Figure 10-4: Simple Aggregation Example . . . 238
Figure 10-5: Results of Running Example 10.3 . . . 239
Figure 10-6: Composite Aggregation Example . . . 239
Figure 10-7: Results of Running Example 10.6 . . . 240
Figure 10-8: Sequence Diagram – Simple Aggregation . . . 240
Figure 10-9: Sequence Diagram – Composite Aggregation . . . 241
Figure 10-10: Engine Simulation Project Specification . . . 242
Figure 10-11: Engine Simulation Class Diagram . . . 243
Figure 10-12: Engine Class Diagram . . . 243
Figure 10-13: Create Engine Object Sequence . . . 245
Figure 10-14: Result of Running Example 10.7 . . . 246
Figure 10-15: Simple Aggregation Class Diagram . . . 251
Figure 10-16: Composite Aggregation Class Diagram . . . 251
Figure 11-1: Inheritance Hierarchy Illustrating Generalized and Specialized Behavior . . . 256
Figure 11-2: UML Class Diagram Showing DerivedClass Inheriting from BaseClass . . . 258
Figure 11-3: UML Diagram of BaseClass and DerivedClass Showing Fields, Properties, and Methods . . . 259
Figure 11-4: Results of Running Example 11.3 . . . 261
Figure 11-5: UML Diagram Showing Student Class Inheritance Hierarchy . . . 262
Figure 11-6: Results of Running Example 11.6 . . . 264
Figure 11-7: Results of Running Example 11.7 . . . 265
Figure 11-8: UML Class Diagram For BaseClass & DerivedClass . . . 266
Figure 11-9: Results of Running Example 11.3 with Modified Versions of BaseClass and DerivedClass . . . 267
Figure 11-10: Expressing an Abstract Class in the UML . . . 268
Figure 11-11: UML Class Diagram Showing the AbstractClass and DerivedClass Inheritance Hierarchy . . . 269
Figure 11-12: Results of Running Example 11.12 . . . 270
Figure 11-13: Two Types of UML Interface Diagrams . . . 271
Figure 11-14: UML Diagram Showing the Simple Form of Realization . . . 272
Figure 11-15: UML Diagram Showing the Expanded Form of Realization . . . 272
Figure 11-16: UML Diagram Showing the MessagePrinter Class Implementing the IMessagePrinter Interface . . . 272
Figure 11-17: Results of Running Example 11.15 . . . 273
Figure 11-18: Employee Class Inheritance Hierarchy . . . 276
Figure 11-19: Results of Running Example 11.20 . . . 278
Figure 11-20: Engine Simulation UML Class Diagram . . . 279
Figure 11-21: Results of Running the EngineTestApp . . . 280
Figure 12-1: Form Class Inheritance Hierarchy . . . 292
Figure 12-2: Results of Running Example 12.1 . . . 293
Figure 12-3: A Standard Window can be Resized by Dragging the Lower Right Corner . . . 294
Figure 12-4: Windows Message Routing (Message Pump) . . . 295
Figure 12-5: Results of Running Example 12.2 . . . 296
Figure 12-6: Screen Coordinate System . . . 297
Figure 12-7: Window Coordinates . . . 298
Figure 12-8: Results of Running Example 12.3 . . . 299
Figure 12-9: Running Example 12.4 via the Command Line with the Name of the Image WCC_2.jpg . . . 300
Figure 12-10: Running Example 12.4 with no Image . . . 301
Figure 12-11: Results of Running Example 12.5 . . . 302
Figure 12-12: Results of Running Example 12.6 with Different Text in the TextBox . . . 305
Figure 12-13: UML Class Diagram Showing Separate GUI and Application/Event Handler Classes . . . 306
Figure 12-14: Results of Running Example 12.8 – GUI Events Handled in Separate Object . . . 308
Figure 12-15: Results of Running Example 12.10 – Buttons Adjust when Window is Resized . . . 310
Figure 12-16: Results of Running Example 12.12 after several Buttons have been Clicked . . . 311

© 2008 Rick Miller — All Rights Reserved

Figure 12-17: Window and Menu Structure of Menu Demo Program 312
Figure 12-18: Results of Running Example 12.14 and Adding Several Buttons and Text Boxes 314
Figure 12-19: Results of Running Example 12.16 – Double-clicking the First Line 316
Figure 13-1: Event Publisher and Subscriber 322
Figure 13-2: Event Publisher and Subscriber 323
Figure 13-3: Minute Tick UML Class Diagram 324
Figure 13-4: Results of Running Example 13.5 325
Figure 13-5: Water Tank System UML Class Diagram 326
Figure 13-6: Results of Running Example 13.11 332
Figure 14-1: Results of Testing DynamicArray 340
Figure 14-2: Results of Running Example 14.3 341
Figure 14-3: Results of Running Example 14.4 341
Figure 14-4: Array of Object References Before Insertion 342
Figure 14-5: New Reference to be Inserted at Array Element 3 (index 2) 342
Figure 14-6: Array After New Reference Insertion 343
Figure 14-7: Linked List Node Organization 343
Figure 14-8: Linked List Before New Element Insertion 344
Figure 14-9: New Reference Being Inserted Into Second Element Position 344
Figure 14-10: References of Previous, New, and Next List Elements must be Manipulated 344
Figure 14-11: Linked List Insertion Complete 345
Figure 14-12: A Hash Function Transforms a Key Value into an Array Index 345
Figure 14-13: Hash Table Collisions are Resolved by Linking Nodes Together 346
Figure 14-14: Red-Black Tree Node Data Elements 346
Figure 14-15: Red-Black Tree After Inserting Integer Values 9, 3, 5, 6, 7, 8, 4, 1 346
Figure 14-16: A Stack After Several Push and Pop Operations 347
Figure 14-17: A Queue After Several Enqueue and Dequeue Operations 347
Figure 14-18: Results of Running Example 14.6 352
Figure 14-19: Results of Running Example 14.8 353
Figure 14-20: Results of Running Example 14.9 355
Figure 14-21: Results of Running Example 14.11 356
Figure 14-22: Results of Running Example 14.13 359
Figure 14-23: Results of Running Example 14.15 360
Figure 14-24: Results of Running Example 14.16 361
Figure 15-1: Exception Information Table 367
Figure 15-2: Exception Class Hierarchy 368
Figure 15-3: Getting Exception Information from MSDN 369
Figure 15-4: Results of Running Example 15.1 371
Figure 15-5: Results of Running Example 15.2 372
Figure 15-6: Results of Running Example 15.3 373
Figure 15-7: Results of Running Example 15.4 373
Figure 15-8: Results of Running Example 15.7 376
Figure 16-1: List of Running Applications 383
Figure 16-2: Partial List of Processes Running on the Same Computer 383
Figure 16-3: Processes and their Threads Executing in a Single-Processor Environment 384
Figure 16-4: Processes and their Threads Executing in a Multiprocessor Environment 385
Figure 16-5: SingleThreadedVacation Program Output 387
Figure 16-6: MultiThreadedVacation Program Output - Partial Listing 388
Figure 16-7: Thread States and Transition Initiators 389
Figure 16-8: Results of Running Example 16.3 390
Figure 16-9: Results of Running Example 16.4 391
Figure 16-10: Results of Running Example 16.5 392
Figure 16-11: Results of Running Example 16.6 393
Figure 16-12: Results of Running Example 16.7 393
Figure 16-13: Results of Running Example 16.8 394
Figure 16-14: Results of Running Example 16.9 395
Figure 16-15: One Particular Result of Running Example 16.10 398
Figure 16-16: Partial Result of Running Example 16.11 400
Figure 16-17: Results of Running Example 16.12 401
Figure 16-18: Results of Running Example 16.13 402

© 2008 Rick Miller — All Rights Reserved

Figure 16-19: Results of Running Example 16.14 403
Figure 17-1: Simplified View of Service Layers 410
Figure 17-2: Typical Directory Structure 411
Figure 17-3: The Absolute Path to the Reports\East\Q2.xls File 411
Figure 17-4: Results of Running Example 17.1 412
Figure 17-5: Results of Running Example 17.3 416
Figure 17-6: Results of Running Example 17.4 417
Figure 17-7: Results of Running Example 17.6 420
Figure 17-8: Results of Running Example 17.8 422
Figure 17-9: Legacy Datafile Adapter Project Specification 423
Figure 17-10: Header and Record Length Analysis 424
Figure 17-11: Monitor.Enter()/Monitor.Exit() vs. the lock Keyword 425
Figure 17-12: Results of Running Example 17.16 Once 437
Figure 17-13: Results of Running Example 17.19 440
Figure 17-14: Results of Running Example 17.21 and Selecting Three Files 442
Figure 18-1: A Simple Computer Network 450
Figure 18-2: Local Area Network Connected to the Internet 451
Figure 18-3: The Internet – A Network of Networks Communicating via Internet Protocols 452
Figure 18-4: Client and Server Hardware and Applications 453
Figure 18-5: Client and Server Applications Physically Deployed to the Same Computer 454
Figure 18-6: Running Multiple Clients on Same Hardware 455
Figure 18-7: Client and Server Applications Deployed on Different Computers 455
Figure 18-8: A Multitiered Application 456
Figure 18-9: Physically Deploying Logical Application Tiers on Same Computer 457
Figure 18-10: Logical Application Tiers Physically Deployed to Different Computers 457
Figure 18-11: TCP/IP Protocol Stack 458
Figure 18-12: Internet Protocol Stack Operations 459
Figure 19-1: .NET Remoting Architecture 466
Figure 19-2: RemotingServer Waiting for Something to do 468
Figure 19-3: Results of Running RemotingServer and RemotingClient with a SingleCall Mode Remote Object 469
Figure 19-4: Results of Hosting TestClass Remote Object in Singleton Mode 470
Figure 19-5: Results of Accessing a Remote Object via an Interface 472
Figure 19-6: Results of Running RemotingServer and RemotingClient with Configuration Files 473
Figure 19-7: Results of Sending a Collection of Person Objects to a Remoting Client 477
Figure 19-8: Server Application Listens on a Host and Port for Incoming TcpClient Connections 478
Figure 19-9: TcpListener Accepts Incoming TcpClient Connection 478
Figure 19-10: TcpClients Communicate via a NetworkStream using StreamReader and StreamWriter Objects 479
Figure 19-11: Results of Running the EchoClient and EchoServer Applications 481
Figure 19-12: Two Clients Connected to MultiThreadedClientServer 482
Figure 19-13: Results of Running MultiIPEchoServer and EchoClient (Mod 1) Applications 485
Figure 19-14: Results of Running SurrealistEchoServer and EchoClient (Mod 2) 489
Figure 20-1: Employee Training Server Application Architecture 494
Figure 20-2: SQL Server System Configuration Check 496
Figure 20-3: SQL Express Feature Selection Dialog 496
Figure 20-4: Results of Testing SQL Server Express Edition Installation 497
Figure 20-5: Management Studio Login Dialog 497
Figure 20-6: SQL Management Studio Main Window 497
Figure 20-7: Enterprise Library Custom Setup Dialog 498
Figure 20-8: Double-Click the InstallServices.bat File 498
Figure 20-9: Enterprise Library Configuration File Creation Tool 500
Figure 20-10: Contents of the SimpleConnection Project Directory Before Compiling 500
Figure 20-11: Results of Running the SimpleConnection Application 501
Figure 20-12: The Primary Key of One Table Can Serve as the Foreign Key in a Related Table 501
Figure 20-13: SQL Server's Default Databases 502
Figure 20-14: Creating EmployeeTraining Database with SQL Command Utility 503
Figure 20-15: Checking on the Existence of the EmployeeTraining Database 503
Figure 20-16: Results of Executing the create_database.sql Script 504
Figure 20-17: Results of Executing create_tables.sql Database Script 505
Figure 20-18: Results of Running create_test_data.sql Database Script 507

© 2008 Rick Miller — All Rights Reserved

Figure 20-19: Results of Executing a Simple Select Statement 508
Figure 20-20: Selecting Specific Rows with select Statement 508
Figure 20-21: Inserting More Test Data with the create_test_data.sql Database Script 509
Figure 20-22: Results of Limiting Data Returned from select Statement with where Clause 509
Figure 20-23: Results of Executing the Previous Query 509
Figure 20-24: Changing Coralie Powell's Last Name to Miller with the Update Statement 510
Figure 20-25: Deleting all Employees whose Last Names = "Miller" 510
Figure 20-26: Verifying the Creation of the tbl_employee_training Table 512
Figure 20-27: Selecting EmployeeIDs from tbl_employee 513
Figure 20-28: Results of Running the Previous SQL Query 514
Figure 20-29: Results of Running the Previous SQL Query 515
Figure 20-30: Results of Executing a Cascade Delete and Checking the Results 515
Figure 20-31: Employee Training Project Folder Arrangement 516
Figure 20-32: EmployeeVO and EmployeeDAO Class Diagram 520
Figure 20-33: Results of Running the CompileVO Target using the MSBuild Utility 522
Figure 20-34: Build Warnings From Conflicting Type Declarations 523
Figure 20-35: Initial State of the EmployeeTrainingServer Application Window 530
Figure 20-36: Employee Picture Loaded and Create Button Enabled 531
Figure 20-37: Testing with More Employee Pictures 531
Figure 20-38: Testing the Insertion and Retrieval of a Large Image 532
Figure 20-39: TrainingDAO and TrainingVO Class Diagram 533
Figure 20-40: EmployeeAdminBO UML Class Diagram 533
Figure 20-41: Collapsed Code Regions in Notepad++ 542
Figure 20-42: Modified Test Application 544
Figure 20-43: EmployeeTrainingRemoteObject UML Class Diagram 552
Figure 20-44: EmployeeTrainingServer Running and Ready For Remote Connections 556
Figure 20-45: Client Project Directory Structure 557
Figure 20-46: Running Client Application via the MSBuild Project's Run Target 559
Figure 20-47: EmployeeTrainingClient UML Class Diagram 559
Figure 20-48: Mock-up Sketch of the EmployeeTrainingApplication GUI 560
Figure 20-49: EmployeeTrainingClient Initial Display on Startup – Something's Not Quite Right! 562
Figure 20-50: Employee's Related Training Shown in Training DataGridView 563
Figure 20-51: Results of Clicking on a Employee with a Picture - a RemotingException is Thrown 563
Figure 20-52: Bitmap Class Usage Note 564
Figure 20-53: EmployeeTrainingClient Application with Employee's Picture Displayed in the PictureBox 569
Figure 20-54: Employee Form Mock-up 570
Figure 20-55: Training Form Mock-up 573
Figure 20-56: Main Application Window with Edit Menu Open to Reveal Revised Menu Structure 581
Figure 20-57: Edit Menu Items Disabled 582
Figure 20-58: Empty Employee Data Entry Form 582
Figure 20-59: Employee Form Fully Populate and Submit Button Enabled 582
Figure 20-60: Training Form Empty and Filled 583
Figure 21-1: Method Signature for Overloaded Unary Operator 591
Figure 21-2: Method Signature for Overloaded Unary Logical Operator 591
Figure 21-3: Results of Running Example 21.2 592
Figure 21-4: Results of Running Example 21.4 593
Figure 21-5: Results of Running Example 21.6 594
Figure 21-6: Results of Running Example 21.9 596
Figure 21-7: Overloaded Binary + Operator Signature that Operates on Two Objects of Type MyType 597
Figure 21-8: Overloaded Binary + Operator Signature that Operates on Objects of MyType and Integer 597
Figure 21-9: Results of Running Example 21.11 599
Figure 21-10: Results of Running Example 21.13 601
Figure 21-11: Results of Running Example 21.15 603
Figure 21-12: Method Signature for Overloaded Equality Operator 604
Figure 21-13: Compiler Warning – == and != Operators Need Special Attention 606
Figure 21-14: Results of Running Example 21.17 606
Figure 21-15: Method Signatures for Implicit and Explicit Cast Operators 607
Figure 21-16: Results of Running Example 21.19 609
Figure 21-17: Results of Running Example 21.20 610

© 2008 Rick Miller — All Rights Reserved

Figure 22-1: Horizontal and Vertical Member Accessibility 617
Figure 22-2: Running Example 22.2 Several Times 620
Figure 22-3: Running MainApp in the Read Mode 623
Figure 22-4: Results of Running MainApp Several More Times in the Append Mode then Read Mode 623
Figure 22-5: Concept of a Shallow Copy 625
Figure 22-6: Concept of a Deep Copy 625
Figure 22-7: Results of Running Example 22.6 627
Figure 22-8: Results of Running Example 22.8 629
Figure 22-9: Results of Running Example 22.10 633
Figure 22-10: Results of Running Example 22.12 636
Figure 22-11: Results of Running Example 22.14 637
Figure 23-1: Results of Running Example 23.2 646
Figure 23-2: Results of Running Example 23.4 648
Figure 23-3: Results of Running Example 23.6 650
Figure 23-4: Results of Running Example 24.8 652
Figure 23-5: Strong vs. Weak Types 653
Figure 23-6: Results of Running Example 24.12 655
Figure 23-7: Naval Fleet Class Inheritance Hierarchy 657
Figure 23-8: Results of Running Example 24.22 661
Figure 23-9: Traditional Top-Down Functional Dependencies 662
Figure 24-1: Meyer's Inheritance Taxonomy 671
Figure 24-2: Person-Employee Inheritance Diagram 673
Figure 24-3: Revised Person - Employee Example 677
Figure 24-4: Results of Running Example 24.9 682
Figure 25-1: Results of Running Example 25.4 692
Figure 25-2: Results of Running Example 25.8 695
Figure 25-3: Model-View-Controller Pattern 695
Figure 25-4: Results of Running Example 25.11 and Clicking the "Next Message" Button Several Times 697
Figure 25-5: EmployeeMVC Project Directory Structure 702
Figure 25-6: Interacting with the Employee Management Application 722

© 2008 Rick Miller — All Rights Reserved

Preface

Welcome – And Thank You!

Welcome to *C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming*. Thank you for supporting the writing efforts of a independent author and small publisher. I spent two years of my life crafting this book. My goal was to create a book with no spelling mistakes or typographical errors, a book whose programming examples are complete and actually compile, a book that serves equally well both novice and practitioner, a book that stretches your brain with in-depth material and challenging projects. Above all, I wanted to create a book that gives you the absolute best value for your money. I hope I've achieved my goal. I tried my best and that's all one can do.

Target Audience

C# 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 C# 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 GUI programming techniques, network programming, database access, and object-oriented theory will enable you to take your skills to a higher level.

Approach(es)

The .NET Framework 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 *C# For Artists* include the following:

Say what I'm gonna say; say it; then say what I 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 I think something is especially important I will present it to you several different ways in different chapters in the book.

Illustrate, illustrate, illustrate: Pictures are worth a thousand words. To this end I illustrate difficult concepts graphically whenever 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

©2008 Rick Miller — All Rights Reserved

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 C# text book. This approach is necessary to push you past the stage of simplistic C# programming.

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

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.

Pedagogy – I Mean, How This Book's Arranged

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 capabilities in a particular area. They could be considered focused micro-projects.

©2008 Rick Miller — All Rights Reserved

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.

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 14 point normal with lines above and below.

Ordinary paragraphs like this one are set in Roman font 10 point normal

This Is An Example Of A First Level Subheading

It is set in Peynot font 12 point normal.

This Is An Example Of A Second Level Subheading

It is set in Peynot font 10 point oblique.

This Is An Example Of A Third Level Subheading

It is set in Peynot font 9 point oblique underline 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 font 9 point. 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
```


©2008 Rick Miller — All Rights Reserved

SupportSite™ Website

The *C# For Artists* SupportSite™ is located at [http://pulpfreepress.com/content/SupportSites/CSharpForArtists]. The support site includes source code arranged by chapter plus any corrections or updates to the text.

Problem Reporting

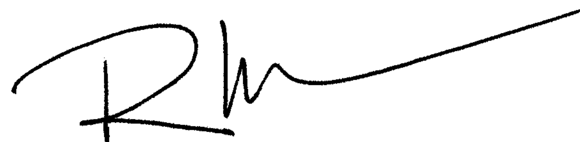
Although I made every possible effort to produce a work of superior quality, some mistakes will no doubt go undetected. All typos, misspellings, inconsistencies or other problems found in *C# For Artists* are mine and mine alone. To report a problem or issue with the text please contact me directly at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

About The Author

Presently, I'm a senior computer scientist and web applications architect for Science Applications International Corporation (SAIC) where I design and build enterprise web applications for the Department of Defense intelligence community. I hold a master's degree in computer science from California State University, Long Beach and am an assistant professor at Northern Virginia Community College, Annandale Campus, where I teach a variety of computer programming courses. I enjoy reading, writing, photography, and celestial navigation. You can view a small sample of my photos at www.warrenworks.com.

Acknowledgments

A book of this size could not have been completed without the help of many people. I'd like to thank my students for providing invaluable feedback on various portions of the text, especially Sebastian Gerke, whose insightful observations and comments prevented me from falling on my sword on several occasions. I'd especially like to praise the efforts of Adrienne Denise Millican for patiently reviewing and editing large portions of the book. And finally, I want to acknowledge the love and support of family and friends who always seem to understand when I can't make an appearance because I'm devoted to my craft.



Rick Miller

Falls Church, Virginia

©2008 Rick Miller — All Rights Reserved

Part I: The C# Student Survival Guide

©2008 Rick Miller — All Rights Reserved

©2008 Rick Miller — All Rights Reserved

Chapter 1

Voigtlander Bessa-L / 15mm Super Wide-Heliar

Rosslyn, VA

An Approach To The Art Of Programming

Learning Objectives

- *Describe the difficulties you will encounter in your quest to become a C# 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 phrases "maximize cohesion" and "minimize coupling"*
- *Describe the differences between functional decomposition and object-oriented design*
- *State the meaning of the term "isomorphic mapping"*

© 2008 Rick Miller — All Rights Reserved

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 programming concepts. 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 C#

During your studies of the C# 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 C#. 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 C# 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 C# 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 your choice (*i.e.*, a computer running Microsoft Windows XP, or later, operating system.)
- Problem solving skills
- Project approach techniques
- Project complexity management techniques
- The ability to put yourself in the mood to program
- The ability to stimulate your creativity
- Object-oriented analysis and design
- Object-oriented programming principles
- Microsoft .NET Framework Application Programming Interface (.NET 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 C# 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!

© 2008 Rick Miller — All Rights Reserved

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", 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 still 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.

Personality Traits Found In Great Programmers

Software engineers come in all shapes, sizes, and temperaments. I've worked with many over the years. Here I'd like to discuss what I believe are a few of the most important personality traits shared by the best. I'm not trying to describe the perfect person; we all have our strengths and weaknesses. But by observing some really smart people in action, I have formulated a definite opinion regarding the traits they possess that enable them to work well by themselves while at the same time permitting them to perform well in a team environment.

Creative

Like I said at the beginning of the chapter, the most prevalent personality trait great programmers possess is that of creativity. Solving problems in such a manner that allows them to be executed by a machine takes truck loads of creativity.

If you say to yourself, "But I'm not creative!" My advice to you is not to sell yourself short. A large part of being creative is simply having an open mind. You must be receptive to alternative solutions and not limit yourself to a "this way or the highway" way of thinking.

Tenacious

Great programmers never give up! As computers, operating systems, and programming languages grow increasingly complex, so too grows the complexity of their associated development environments and the range of issues and problems you will encounter when developing solutions for these machines. If you are the type of person who likes to bite into a problem like a pit bull and keep at it until you've licked it, then you'll do well as a programmer.

Resilient

Great programmers bounce back! When a particular problem has given you a thorough trouncing you must come back strong the next day and fight the battle again. Programming is one continuous stream of problem solving. This you must be willing to repeat ad-infinitum. To paraphrase an old Timex® watch advertisement campaign slogan, you must be able to "...take a licking and keep on ticking!"

Methodical

Great programmers approach everything they do in a methodical way. This holds true regardless of if you program alone or as part of a team or if a formal methodology does or does not exist. You must be able to formulate problem attack plans and execute those plans.

© 2008 Rick Miller — All Rights Reserved

Meticulous

Great programmers are meticulous. Close attention to detail is paramount in the programming profession. One identifier misspelled, one token out of place, can break entire systems.

Honest

Great programmers can be trusted to do the right thing in the code when no one is looking. They must be honest with themselves but especially towards other programmers. Honest programmers put in an honest day's work and give realistic estimates regarding task completion.

Proactive

Great programmers recognize and capitalize upon opportunity. They get up out of their chair and go out and talk to their fellow programmers. When they see problems in the code or areas for improvement they bring it to the attention of the team.

Humble

Great programmers know when to seek guidance or help. They don't let their ego stand in the way of the greater good. They get up off their duff and talk to their fellow programmers. They share their knowledge and wisdom so that someday they can take a vacation. Most importantly, admitting that they don't know something early on can save hundreds of wasted work hours down the line.

Be a Generalist and a Just-in-Time Specialist

Great programmers are well-versed in all aspects of computing. Rarely have I ever met any who referred to themselves as only a this type of programmer or a that type of programmer. I'd rather hire generalists with solid educational backgrounds and the proven ability to teach themselves new tricks, than to bank on a specialist who refuses to grow professionally. In other words, great programmers have a broad range of skills they can apply to the problem. Great programmers can gather requirements, design a solution, write the code, conduct testing, write supporting documentation, deploy the application if necessary, and carry on intelligent conversations with the customer to boot.

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 programming 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.

© 2008 Rick Miller — All Rights Reserved

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 leaving 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 your instructor and asking them 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. (*See the discussion of the Quality without a Name (QWAN) in Chapter 25 — Helpful Design Patterns*)

Two objectives of good software design are the abilities to *accommodate change* and *tame complexity*. Accommodating 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 good organization upon your source code files. For simple projects you can group source code files together in one directory. For more complex projects you will want to organize source code files into subfolders and group related type definitions into *namespaces*. (*See Chapter 20 — Database Access*)

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

© 2008 Rick Miller — All Rights Reserved

for every project. After you have used it a few times to get 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 verify what I believed an instructor required by asking them to clarify any points I did not understand.

Problem Domain

The problem domain is the body of knowledge necessary to implement a software solution apart and distinct from the knowledge of programming itself. For example, consider the following application requirement: "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 for what you are solving such 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 programming students at the opening stages of the project is knowing what solution to design without knowing enough of the programming 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. 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. Then 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 identifies and implements program data structures separately from the program code that manipulates those data structures. When taking a procedural-based approach to a solution you generally break the problem into small, easily solvable pieces called *functions*, implement the solution to each function separately, and then combine the functions into a complete solution. This methodology is also known as *functional decomposition*.

© 2008 Rick Miller — All Rights Reserved

Although C# does not support standalone functions (*C# has methods and a method must belong to a class*), you can still use a procedural-based design approach to create a working C# program. However, 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. This approach no longer considers data structures and the methods that manipulate those data structures 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. (*Encapsulation*) Data structures and the methods that manipulate them combine to form classes from which objects can then be created.

To solve a programming problem with an object-oriented approach, decompose it into a set of objects and their associated behavior. You can use design tools such as the Unified Modeling Language (UML) to help with this task. Once you've identified system objects, you then define object interface methods. From here you declare classes or structures and implement those interface methods. Finally, you combine these classes or structures 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.

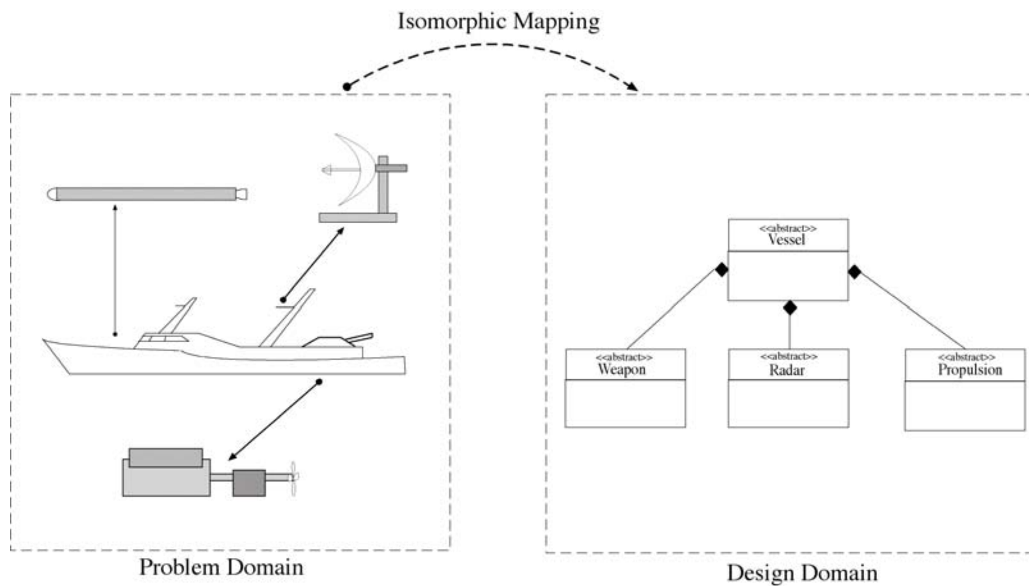


Figure 1-1: Isomorphic Mapping Between Problem Domain and Design Domain

Referring to Figure 1-1 — real world objects such as weapon systems, sensors, propulsion systems, and vessels can have a corresponding representation in the software system design. The correlation between real world objects and software components 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 still implement your design. This means writing class member methods one line of code at a time.

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

 © 2008 Rick Miller — All Rights Reserved

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

The project-approach strategy can be summarized as follows: 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 project-approach strategy presented previously is not intended to replace a formal course on software engineering, but it will help you when you enter the real world as a paid 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 observe 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 in paying 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

© 2008 Rick Miller — All Rights Reserved

had the solution come to you in the middle of the night, 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, own your own computer. It should be one sufficiently powerful to use for C# 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, 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, even 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 for you 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 in the following section, 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 best 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.

© 2008 Rick Miller — All Rights Reserved

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

As with 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

The 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 even a great day. Perhaps your spouse or significant other is being a complete jerk! Perhaps he or she is being especially nice and you're 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

The third stage: 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. The clarity of your purpose takes hold and propels you forward.

Calm and Complete Focus

The fourth stage: You don't notice it at first, but at some point between this stage and the previous stage, you have slipped into a deeply relaxed state. You are utterly focused on the task at hand. It is like becoming completely absorbed in a good book. Someone can call your name, but you will not notice. You will not respond until someone either shouts at you or does 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, leaving you feeling agitated and eager to settle in once again. If you avoid 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 uncover fatal flaws that you must correct if you hope to have a polished, finished project.

 © 2008 Rick Miller — All Rights Reserved

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 both separately and 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 whenever possible. If you discover design flaws or ways to improve the design of your project, you must revise and improve the design to accommodate further development. An example of design refactoring is 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 an iterative 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 this: don't do it! Use the iterative programming cycle previously outlined. Nothing will depress you more than seeing a million compiler errors scroll up the screen after waiting until the bitter end to compile your project.

A Helpful Trick: Stubbing

Use stubbing to both speed development and avoid writing a ton of code just to get something useful to compile. Stubbing is a programming trick that is best illustrated by example.

Suppose 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 first is write and test the menu's display and selection methods before 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!

© 2008 Rick Miller — All Rights Reserved

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 for this is that the first compiler error, 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 other software components. A project's architectural design directly results from the solution approach conceived by one or more software engineers to implement a software solution for 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. The former will think in terms of modules and sub-modules, while the latter will draw a direct correlation to real world objects and their derived software components. The functional decomposition approach will almost always yield software modules that are difficult to use out of context. Modules are so tightly integrated with each other that extracting one for reuse in another system may be impossible. Software architectures based on functional decomposition tend to be brittle and change resistant. By brittle, I mean that a change in one module will have negative effects on other, seemingly unrelated modules. Software based on such change resistant architectures is hard to maintain, modify, or extend.

An understanding of software design patterns will give the object-oriented engineer a double advantage. Such patterns capture the knowledge and experience of many talented software engineers. Their use can significantly increase the flexibility, maintainability, and extensibility of the applications upon which they are based.

However, writing a program in C#, or in any other object-oriented programming language does not automatically 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.

Tame conceptual complexity 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. It's easier for object-oriented programmers to "get their heads

© 2008 Rick Miller — All Rights Reserved

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. 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, UML is more than just pretty pictures. 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. *C# 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.

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 Subversion 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 namespace structure can be applied to 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. It might be easy to compile every class in the directory at the same time. However, the same one-directory organization will simply not work on a large project with teams of programmers creating and maintaining hundreds or thousands of source files.

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 focuses on its intended purpose. *A high degree of 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 external software modules. *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.

© 2008 Rick Miller — All Rights Reserved

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.

Great programmers are creative, tenacious, resilient, methodical, meticulous, honest, proactive, and humble. Great programmers cultivate a broad range of skills and focus on a particular technology when necessary.

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 C# 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.

Skill-Building Exercises

None

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 C# 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 C# reference books listed in the reference section of each chapter in this book.

5. **Web Search:** Conduct a web search for C# and object-oriented programming sites. Bookmark any site you feel

© 2008 Rick Miller — All Rights Reserved

might be helpful to you as you master the C# language. Microsoft's site should be first on your list!

Self-Test Questions

1. List at least seven skills you must master in your studies of the C# 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

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

Notes

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Chapter 2

Nikon F3HP. Kodak Tri-X

Artifacts

Small Victories
Creating C# Projects

Learning Objectives

- List and describe the minimum development tools required to create C# programs
- State the purpose and use of operating system environment variables
- List and describe the steps required to create operating system environment variables
- State the purpose of the Path environment variable
- List and describe the steps required to add the .NET runtime folder path to the Path environment variable
- List and describe the steps required to create C# programs using Microsoft's free command-line tools
- List and describe the development tools found in the Microsoft Windows Software Development Kit (SDK)
- Describe the functions and features generally found in an IDE
- List and describe the steps required to create C# programs using Microsoft Visual C# Express
- Demonstrate your ability to create C# projects using Microsoft's free command-line tools
- Demonstrate your ability to create C# projects using Microsoft's free Visual C# Express

© 2008 Rick Miller — All Rights Reserved

Introduction

I call this chapter Small Victories because selecting development tools and properly configuring your development environment easily accounts for seventy-five percent of the headaches you'll get when starting down the road of C# .NET software development. Creating, compiling, and running your first project represents the biggest hurdle novice programmers face. The information in this chapter is designed to help you easily jump that hurdle and do a victory dance.

Here you will learn several critical software development skills. First, I explain exactly what you need to write, compile, and run C# programs. The good news is that you don't need a fortune to start programming with C# and the .NET Framework. Microsoft offers powerful and flexible software development tools absolutely free. Next, I will explain the purpose and use of environment variables and show you how to configure the PATH environment variable so you can compile and run C# programs from the command console. And, since you may not be familiar with the command console, I will explain its purpose and use and show you how to use several important commands.

Later, once you understand how to use Microsoft's C# .NET command-line tools, I'll show you how to create programs using Microsoft Visual C# Express. Visual C# Express is Microsoft's free integrated development environment (IDE). An IDE increases programmer productivity by providing, under a common user interface, several important software development tools including source code editing, compiling, debugging, and execution.

Creating Projects With Microsoft C#.NET Command-Line Tools

You need only two things to create professional, robust, C# .NET projects: the Microsoft .NET Framework, and a suitable text editor. Both can be obtained free of charge, although you will most likely want to buy a good text editor.

The .NET Framework supplies you with the C# command-line compiler. The C# command-line compiler is a program that is run in a console window and is used to transform programs written in C# into byte-code modules that can be executed by the .NET Common Language Runtime (a.k.a. the .NET virtual machine).

The question most students ask when I teach them how to use these tools is, "Why?". "Why, if Microsoft offers Visual C# Express (or Visual Studio), do I need to know how to create programs using command-line tools?" That's a good question with several answers, and they go something like this: Visual C# Express is a powerful program. In fact, it's so powerful that you can spend a lot of time just learning what it does and how to use it. So, in order to let students focus on learning the C# language, I recommend they learn how to use the command-line tools and postpone their involvement with Visual C# Express until they get some programming experience.

My second answer to the question has a more practical side. Nowadays, most novice programming students have little or no experience using the command console. They are familiar with the Microsoft Windows interface and pointing and clicking with a mouse, but issuing DOS commands from the command line is something completely alien to them.

I consider the ability to compile programs with the C# command-line tool, issuing DOS commands, and setting and using operating system environment variables to be fundamental skills that all programmers need to have in their tool belt. Also, mastering these fundamental skills will let you better understand what Visual C# Express is doing "under the hood". You may find it necessary one day to dive into the code generated "automagically" by Visual C# Express to make a few adjustments. The only way you'll be able to do that is to take complete control of your development environment and understand how to use the command-line tools to compile and run C# programs.

Downloading And Installing The .NET Framework

The first thing you need to do is to download and install the .NET Framework. Microsoft offers the .NET Framework Redistributable Package as a free download from their Microsoft Developer Network website (MSDN) [www.msdn.com]. You can optionally order the .NET Framework on DVD for a nominal charge.

 © 2008 Rick Miller — All Rights Reserved

You will find the .NET Framework 3.5 by going to MSDN and under the *Server Development* heading click on *.NET Framework*. On the *.NET Framework* page under the *Get the Framework* heading click on the *.NET Framework 3.5* link. This takes you to the Microsoft .NET Framework 3.5 download page.

At a minimum, you only need to download the .NET Framework 3.5. It provides the .NET runtime environment and the C# command-line compiler. It also includes the .NET Framework 3.0 Service Pack 1, the .NET Framework 3.0 Redistributable Package, and the .NET Framework 2.0 Service Pack 1. You can optionally download and install the Microsoft Windows Software Development Kit (SDK) for .NET Framework 3.5. The Microsoft Windows SDK provides additional development tools. However, before you can install the SDK, you must download and install the .NET Framework.

Installation of the .NET Framework is straightforward. The important thing to note during the installation process is where on your hard drive the .NET Framework is installed. The path to the .NET Framework installation directory will be [`c:\Windows\Microsoft.NET\Framework\`]. Figure 2-1 shows the .NET Framework directory structure as it appears on my computer.

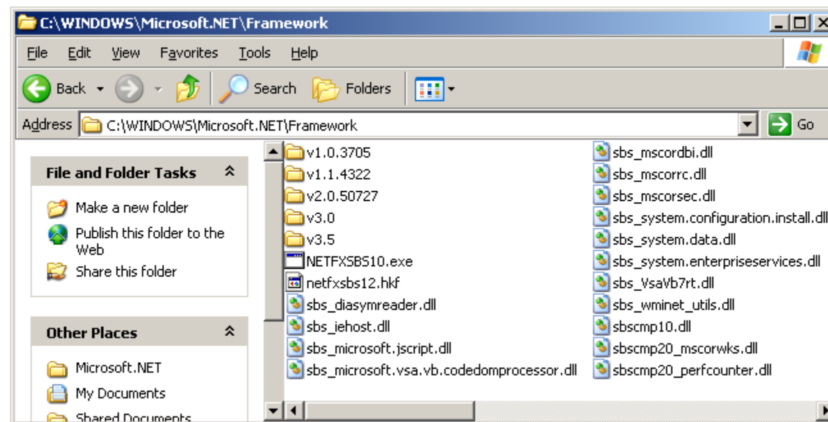


Figure 2-1: Microsoft.NET Framework Installation Directory

Notice in Figure 2-1 the five folders that begin with the letter 'v'. These are the five versions of the .NET Framework installed on my computer. For the purpose of this book we'll only be interested in the most recent version (i.e., the highest numbered folder). Figure 2-2 shows a partial directory listing of the v3.5 folder. In there you'll find the C# compiler command-line tool.

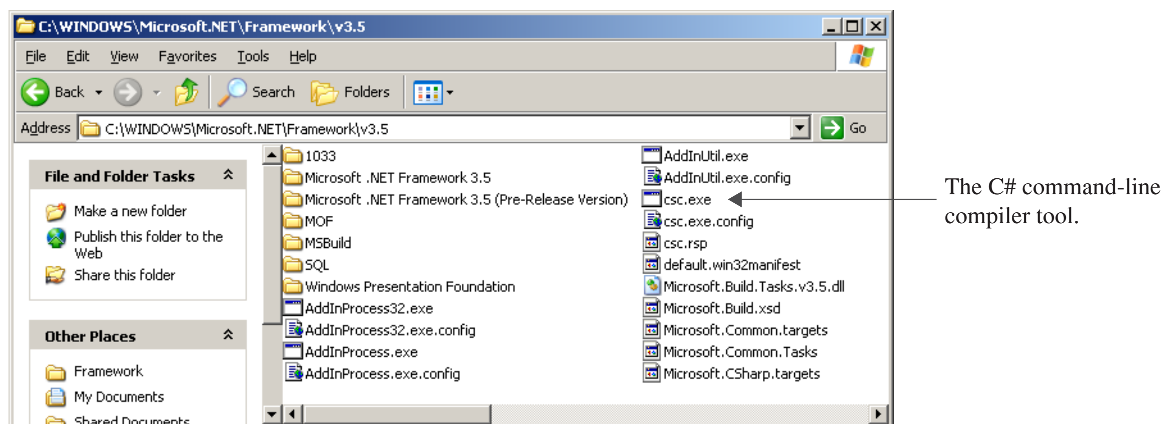


Figure 2-2: Partial Directory Listing of the v3.5 Folder

Now that you've installed the .NET Framework you have everything you need to compile and run C# programs. What you need now is a way to create C# source files. For that you'll need a good text editor. This is the topic of the next section.

© 2008 Rick Miller — All Rights Reserved

Downloading And Installing Notepad++

If you really wanted to rough-it you could use Notepad, the text editor that ships with Microsoft Windows. Notepad is perfectly suitable for small source files and lite editing jobs, but for bigger projects, I recommend getting yourself a copy of Notepad++ from SourceForge.net [http://notepad-plus.sourceforge.net/].

Installation of Notepad++ is quick and straightforward. After you complete the installation a shortcut is automatically installed on your desktop. Although Notepad++ is free to download, install, and use, I recommend that if you like the product and use it often, and if you have the means, that you make a small donation via the website to support its further development.

Armed now with the .NET Framework and a suitable text editor, you have everything you need to create C# programs. But before you get started you'll need to properly configure your development environment.

Configuring Your Development Environment

A properly configured development environment is critical to the software creation process. In this section I will show you how to create and use operating system environment variables, how to create a project folder, how to set folder options so you can see filename suffixes, and how to set-up and configure shortcuts to the command console. The skills you learn in this section will prove time and again to be absolutely invaluable.

Environment Variables

An environment variable is a named location in memory used by Microsoft Windows to store data about the operating system environment. There are generally two types of environment variables: *system* and *user*.

System environment variables store data that pertains to and affects the operating system environment for all users. User environment variables store data that pertains to and affects the operating system environment for a particular user. Some system and user environment variables are automatically created and initialized by the operating system when it is installed and when users are created.

Several important environment variables must be created or edited before you can use the command-line tools to compile and run C# programs. These include: 1) a variable named DOT_NET_FRAMEWORK_HOME that contains the path to the folder location of the most current .NET Framework, and 2) the PATH variable that includes a reference to the DOT_NET_FRAMEWORK_HOME variable so the operating system will know where to find .NET-related executable files like the C# command-line compiler.

Creating Environment Variables

The first environment variable you will set will be the location of the home directory of the .NET Framework. Navigate to that folder now so that you can copy the path to the .NET Framework directory; later, you can paste this value into the environment variable value's text field. (*Doing this prevents you from making mistakes when typing long path names*) The path to this folder should be [`c:\Windows\Microsoft.NET\Framework\v3.5`]. (*Refer to Figure 2-2*) When you open the folder, select the path that shows in the Address box and copy it using CTRL-C.

Next, we'll create the user environment variable named DOT_NET_FRAMEWORK_HOME. See Figure 2-3 for an illustration of the complete environment variable creation process.

Right click the My Computer icon located on your desktop. If this icon is not located on your desktop, click the Start icon located in the lower left part of your screen along the toolbar. Find My Computer and right click it. Click Properties to open the System Properties dialog window. In the System Properties dialog click the Advanced tab, then click the Environment Variables button to open the Environment Variables dialog window. Underneath the User variables section, click the New button to create a new environment variable. This will open the New User Variable dialog window. Enter DOT_NET_FRAMEWORK_HOME into the Variable name textbox. Paste the path to the .NET Framework home directory you copied earlier into the Variable value textbox. After entering both values, your New User Variable dialog window will look similar to the completed example shown in Figure 2-3. Check your work for accuracy, then click the OK button to close the New User Variable dialog window. Click the OK button for each of the remaining open dialog windows to accept the changes. Congratulations! You just created an environment variable.

© 2008 Rick Miller — All Rights Reserved

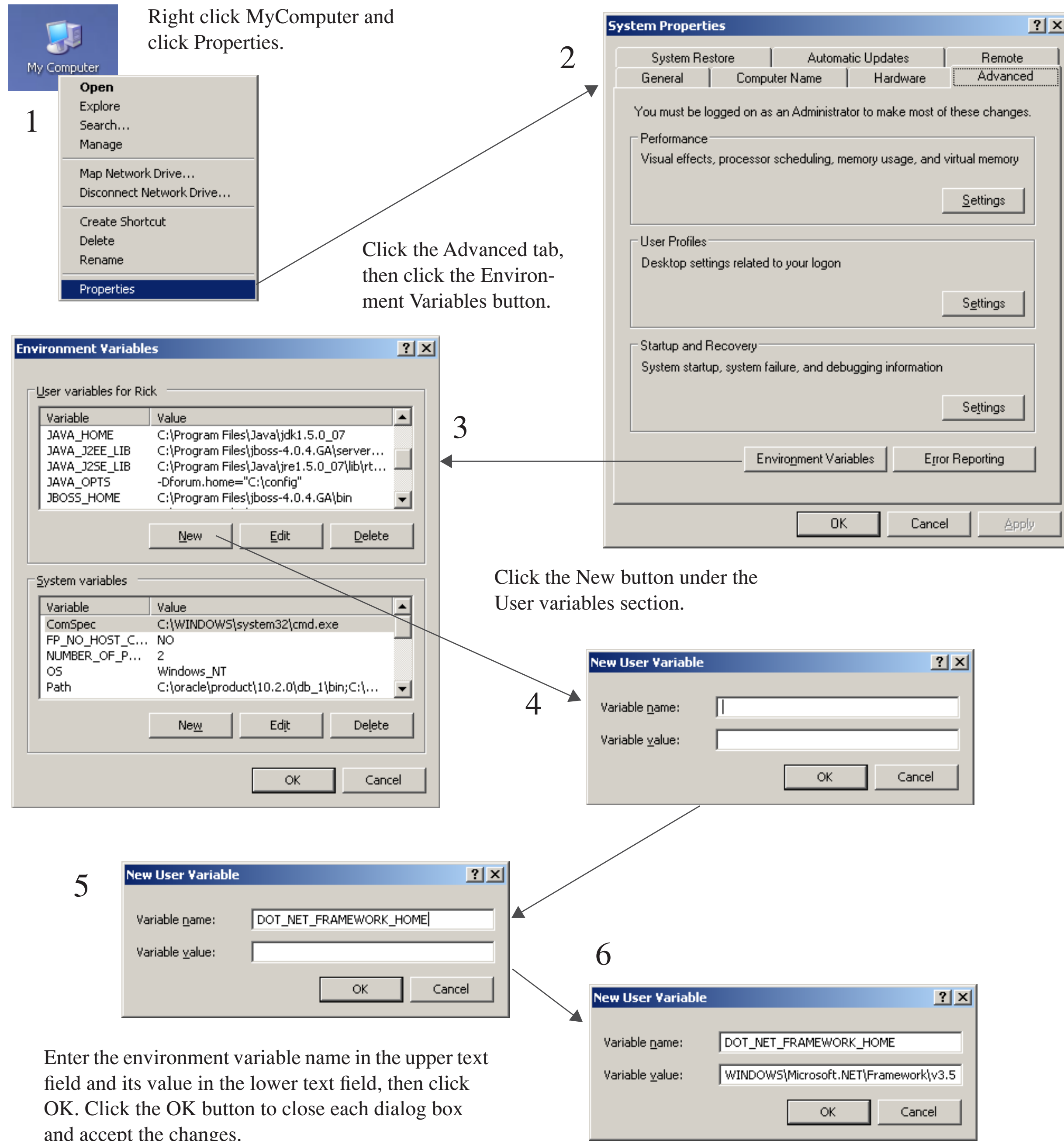


Figure 2-3: Creating an Environment Variable

Creating or Editing the Path Environment Variable

Now that you have created the DOT_NET_FRAMEWORK_HOME environment variable, you can use it to create or edit other environment variables. The next environment variable that you must either create or edit is the Path variable. The operating system uses the Path environment variable to locate executable files. An instance of the Path variable most likely already exists in the System Environment Variables section. I recommend leaving that version alone and creating another Path environment variable in the User Environment Variables section. To create a new User Path environment variable, follow the process illustrated in Figure 2-3. Enter “Path” into the Variable name text

© 2008 Rick Miller — All Rights Reserved

field. Enter the following into the Variable value text field: %DOT_NET_FRAMEWORK_HOME%. Click the OK buttons to accept the changes. (**Note:** To access an environment variable's value, add a "%" to the beginning and end of the variable name.)

If a Path environment variable already exists, you'll need to select it and click the Edit button. Place your cursor in the Variable value text field and move to the far end of the value that's entered in the text field. If that value is not terminated with a semicolon, you'll need to add one before adding the following:

%DOT_NET_FRAMEWORK_HOME%;

Figure 2-4 shows the Path user environment variable being edited on my machine.

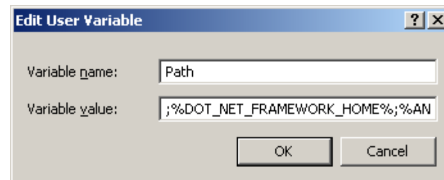


Figure 2-4: Editing the Path User Environment Variable

Testing The Newly Created Environment Variables

Check that you have set your environment variables correctly by running a couple of tests. The first test entails opening a command console and using the DOT_NET_FRAMEWORK_HOME variable in a command. The second test will be running the C# command-line compiler.

First, open the command console. Do this by clicking on Start->All Programs->Accessories->Command Prompt. This will open a command console window like that shown in Figure 2-5. Next, enter the following command at the command prompt: `cd %DOT_NET_FRAMEWORK_HOME%` The `cd` command stands for "Change Directory". If you have set the DOT_NET_FRAMEWORK_HOME environment variable correctly, entering this command should open the .NET Framework v3.5 directory as Figure 2-6 illustrates.

Command prompt

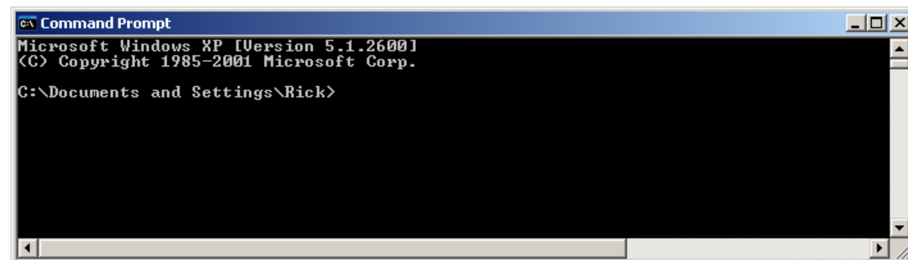


Figure 2-5: Command Console Window

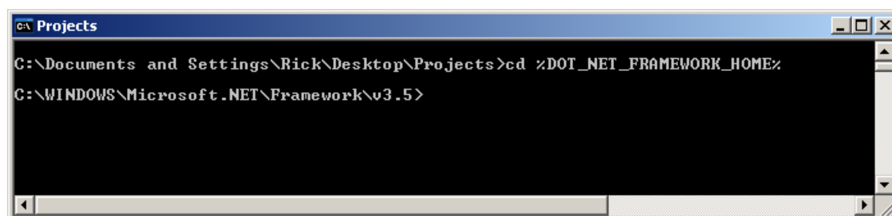


Figure 2-6: Testing the DOT_NET_FRAMEWORK_HOME Environment Variable

Now test the Path environment variable. Execute the following command in a command console window: `csc` This should run the C# compiler which will produce a result similar to that shown in Figure 2-7. If your results look like those shown in Figure 2-7, then all is set properly. Great job! If not, recheck your environment variable settings and try again until you have things set just right.

 © 2008 Rick Miller — All Rights Reserved

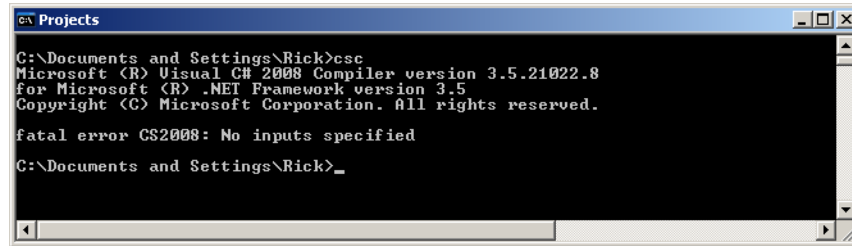


Figure 2-7: Testing the Path Environment Variable by Running the C# Compiler

Creating A Project Folder

The next thing you must do is create a folder in which to store your C# project files. Create a folder named Projects on your C drive or some other acceptable location. This folder will serve as the root folder for any individual projects you create later. You will store each project in its own folder under the Projects folder.

A good place to create the Projects folder is right on your desktop. To do this, right-click your desktop and select New->Folder from the right-click menu as Figure 2-8 illustrates.

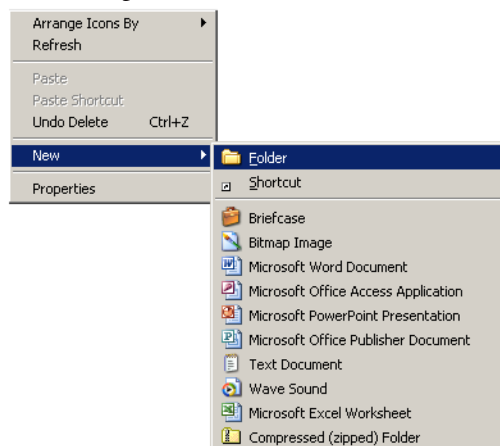


Figure 2-8: Creating a New Folder

Name the new folder “Projects”. Note that when you create a folder on your desktop, you are actually creating it in the [`C:\Documents and Settings\username\Desktop`] folder, where “username” is the username of the account you used to log on to the computer. On my machine, the full path to the Projects folder created on the desktop is this: [`C:\Documents and Settings\rick\Desktop\Projects`]

Setting Folder Options

Now that you have created your Projects folder, you’ll need to change its folder options so you can see file-type extensions. Novice and experienced programmers sometimes have difficulty trying to compile a C# file because the file they thought was saved with an extension of “.cs” was in fact saved with an extension of “.cs.txt”, where the “.txt” extension was automatically added by a text editor, unbeknownst by the programmer. When this happens, the C# compiler will fail to recognize the file as a C# source file. To help prevent such headaches, it’s a great idea to change the folder options of all your folders to show file extensions.

To do this, open the Projects folder and in it create a new text file. The easiest way to create the text file is to simply right click in the open folder and select New->Text Document from the right-click menu. Save the text document with the default name provided. Your Projects folder should now look like Figure 2-9. Notice that the name of the document you just created simply shows as “New Text Document”. The “.txt” extension is hidden by default. So let’s unhide the file extensions. Click the Tools->Folder Options... menu to bring up the Folder Options dialog window. Click the View tab and scroll down until you see the check box that says, “Hide extensions for known file types”. This box is checked by default. Uncheck the box as is shown in Figure 2-10.

© 2008 Rick Miller — All Rights Reserved

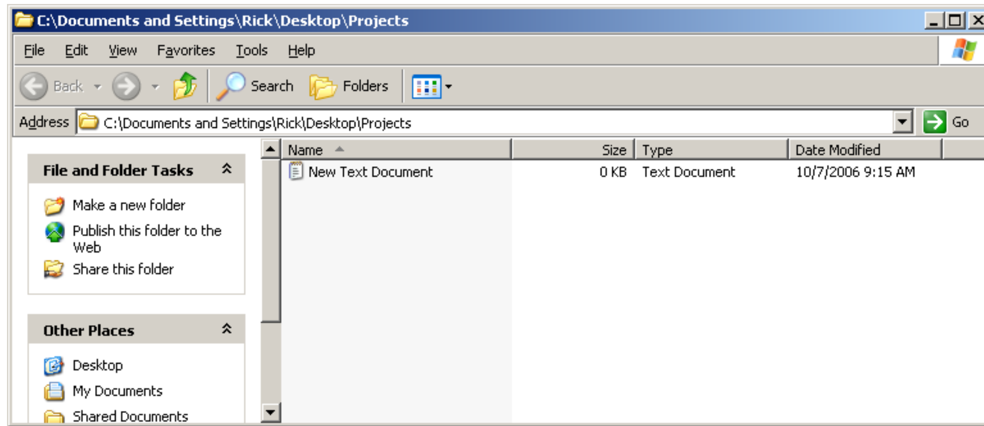


Figure 2-9: Projects Folder Before Setting Folder Options

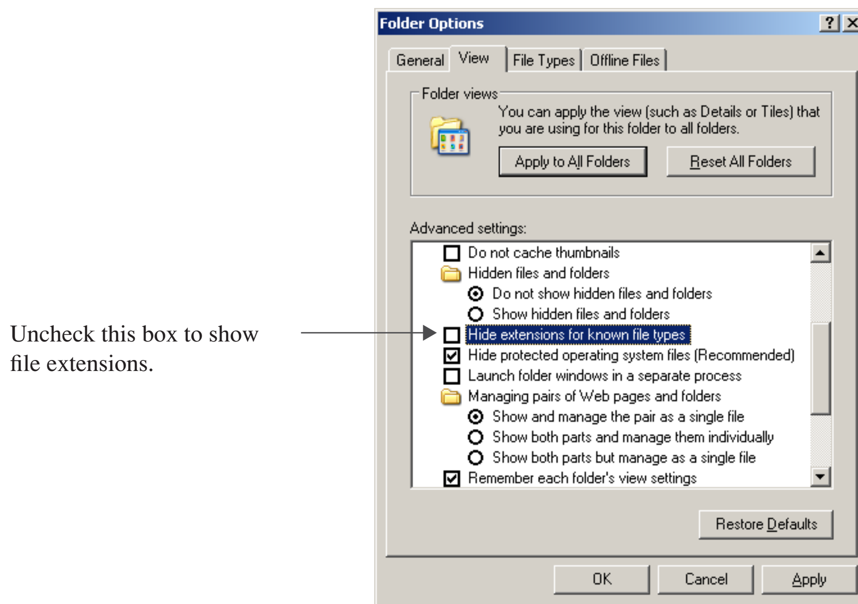


Figure 2-10: Folder Options Dialog Window

Click the Apply to All Folders button in the Folder views section, then click the Apply button and lastly the OK button to dismiss the dialog. Your Projects folder should now look like Figure 2-11. Notice now you can see the ".txt" file extension.

Creating A Shortcut To The Command Console And Setting Its Properties

Since you'll be using the command console to compile C# programs you'll find it convenient to place a command console shortcut on your desktop. To do this, click Start->All Programs->Accessories. Navigate to Command Prompt and right click it. Select Create Shortcut. This will create a new item in the Accessories menu named "Command Prompt (2)". Select the Command Prompt (2) icon and drag it to your desktop. Test the shortcut by double clicking it to open the command console. By default, it should open to the directory `[C:\Documents and Settings\username]`, where "username" is the account you used to log on to the computer. Figure 2-12 shows how the command console window looks with its default settings on my machine.

© 2008 Rick Miller — All Rights Reserved

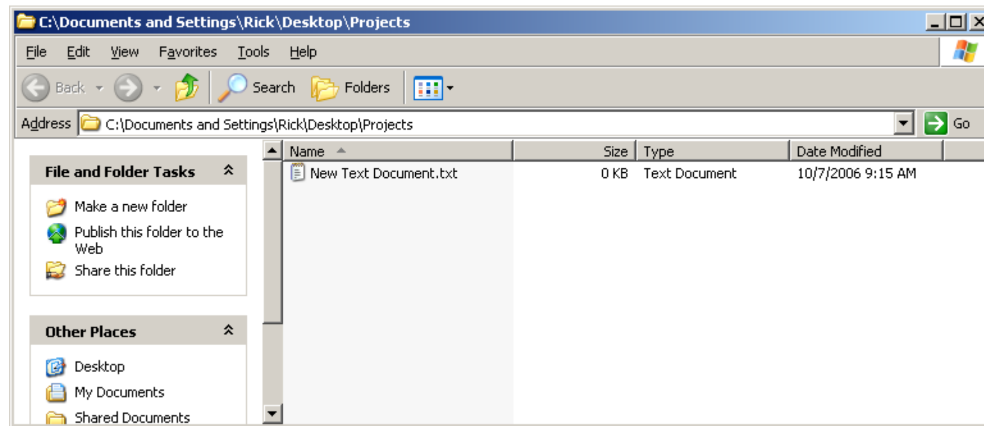


Figure 2-11: Projects Folder After Setting Folder Options

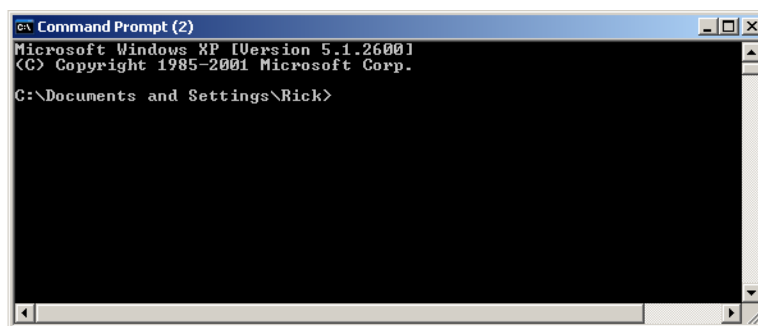


Figure 2-12: Default Command Console Window

Change The Name Of The Command Console Shortcut

The first bit of command console shortcut customization you'll want to do is to change its name. Do this by clicking twice on the shortcut icon's name, pausing between clicks longer than a standard double-click. If you click too fast, you'll simply open the console window. If this happens simply close the window and try again. Rename the shortcut to anything you want, but I recommend changing it to "Projects", or "C# Projects".

Change The Startup Folder Settings

Now that you've changed the command shortcut's name, let's make a more meaningful change. It would be nice if the console window opened automatically in the Projects directory. To make this happen, right click the command console shortcut icon located on your desktop and select Properties. This opens the properties dialog window for that shortcut. For example, if you renamed your shortcut to "Projects", the name of the properties dialog will be "Projects Properties". If you left the name of the shortcut with its default value, the name will be "Command Prompt (2) Properties" as Figure 2-13 illustrates. **Note:** The Shortcut tab is selected by default.

To make the command console automatically start in the Projects directory, change the **Start in** property by replacing its default contents with the full path to the Projects folder. This will be `[C:\Documents and Settings\username\Desktop\Projects\]`, where "username" is the account name you used to log on to the computer. Figure 2-14 shows the command console **Start in** property after I set it on my machine.

Click the OK button to accept the changes. Test the configuration by double clicking the command console shortcut icon. It should open either in the Projects folder, or the folder you designated. If not, recheck your settings and try again until everything works as expected.

 © 2008 Rick Miller — All Rights Reserved

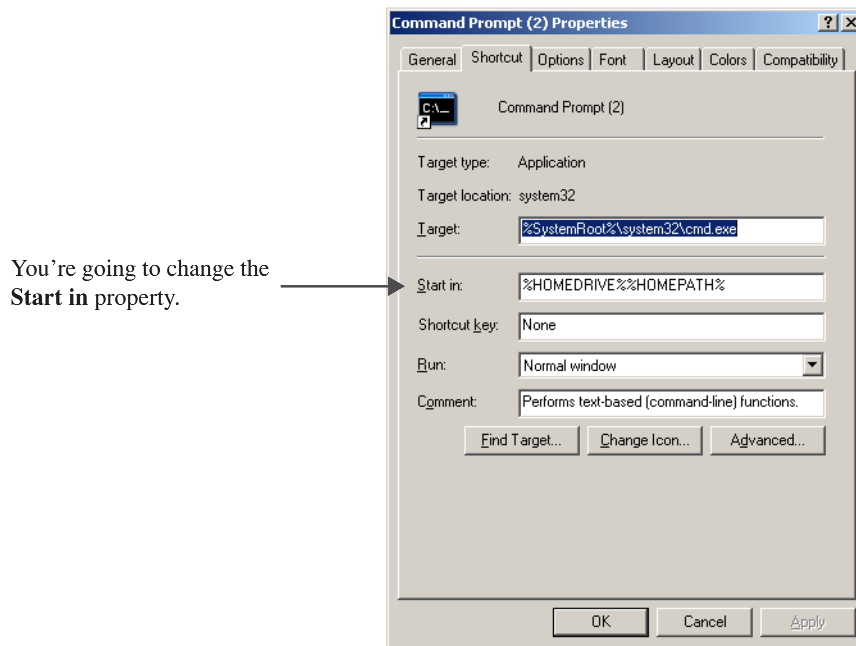


Figure 2-13: Command Console Properties Dialog

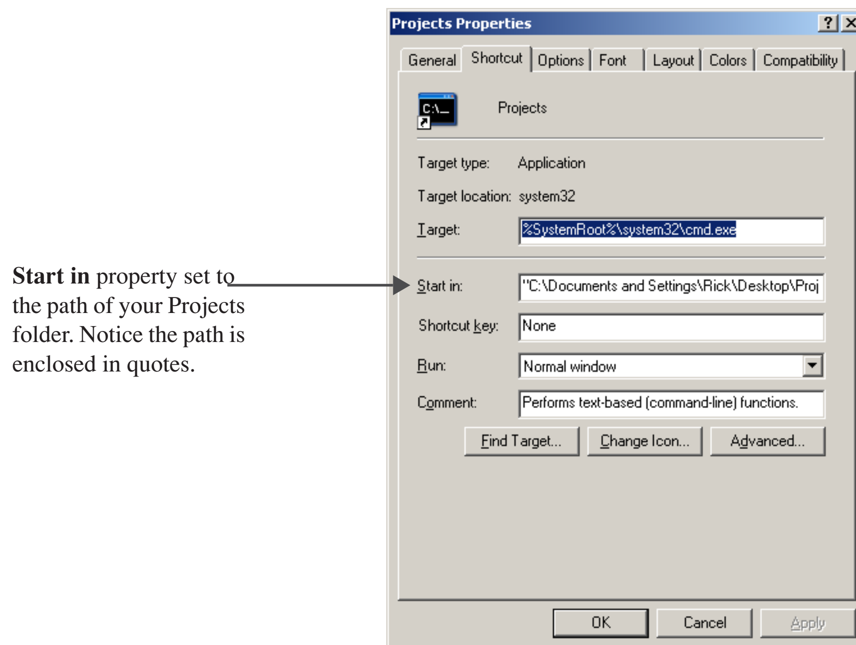


Figure 2-14: Setting the **Start in** Property

Change The Layout Properties

The last adjustment left to make to the command console shortcut is to change its default screen buffer size. This will allow you to increase the length and width of the screen to see more information without the lines wrapping. Once again, right click the command console icon and click the Properties item to open the Properties dialog window. Click the Layout tab and set the screen buffer size **Width** property to 120, and change the **Height** property to 3000, as Figure 2-15 illustrates.

© 2008 Rick Miller — All Rights Reserved

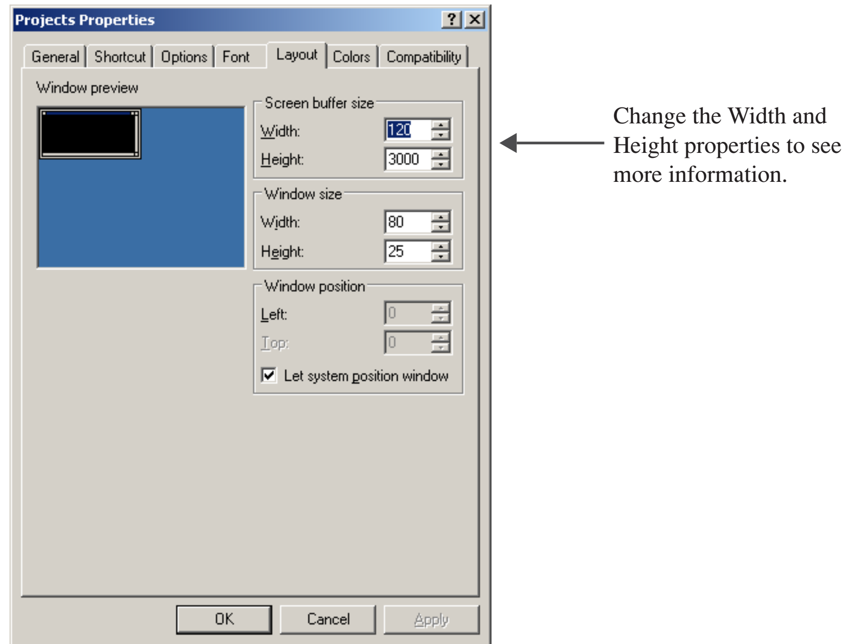


Figure 2-15: Setting Command Console Layout Properties

Click the OK button to accept the new changes. Now, double click the command console shortcut to open the console window. Change its height and width by dragging the lower right-hand corner. You'll find this to be a big help when troubleshooting and debugging your programs.

Testing The Configuration

As a final test of the configuration, let's create and run a short C# program. To do this, you'll need to create the C# source file with the text editor, compile the source file using the C# command-line compiler, and then run the program.

Creating The Source File

Using either Notepad++ or another text editor create a new file named "HelloWorld.cs" and save it in your Projects folder. Enter the code shown in Example 2.1 into your source file and save the file.

2.1 HelloWorld.cs

```
1    using System;
2
3    public class HelloWorld {
4
5      public static void Main(){
6
7         Console.WriteLine("Hello World!");
8       }
9    }
```

Compiling The Source File

To compile the HelloWorld.cs file, open the command console and change to the directory where you saved the file. If you saved it in the Projects folder, then you're already there. If you created a sub folder then change to that directory by using the `cd` command. For example, I saved the file in a folder named "Chapter2" located in the Projects folder. To change to the Chapter2 directory from the Projects directory I entered `cd chapter2`, then pressed the Return or Enter key. Figure 2-16 shows how the console looks on my machine when I use the `dir` command to list the directory contents.

 © 2008 Rick Miller — All Rights Reserved

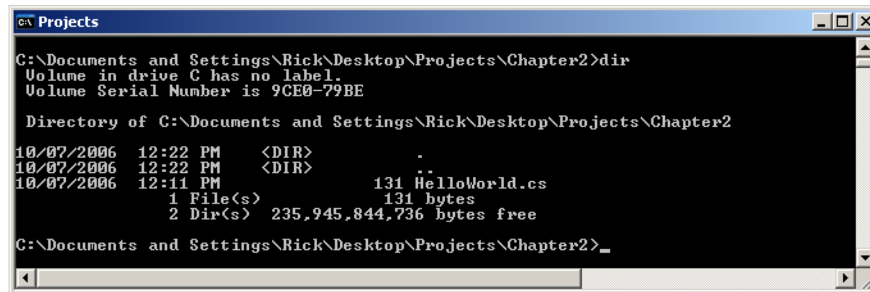


Figure 2-16: Directory Listing of the Chapter2 Directory Showing the HelloWorld.cs File

To compile the HelloWorld.cs file, enter `csc` followed by the name of the source file at the command prompt. If you entered the source code correctly, you should see results similar to those shown in Figure 2-17.

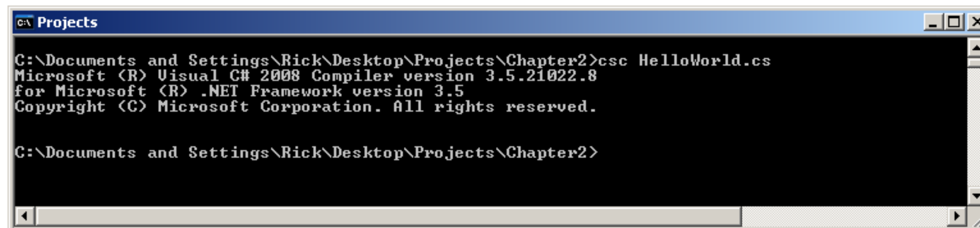


Figure 2-17: Compiling HelloWorld.cs Using the `csc` C# Compiler Command

If you execute another `dir` command to display the directory contents, you'll see a new file named "HelloWorld.exe". This is the executable program file.

Executing The Application

To run the executable file, simply enter its name at the command prompt. Figure 2-18 shows the results of running the Hello World program.

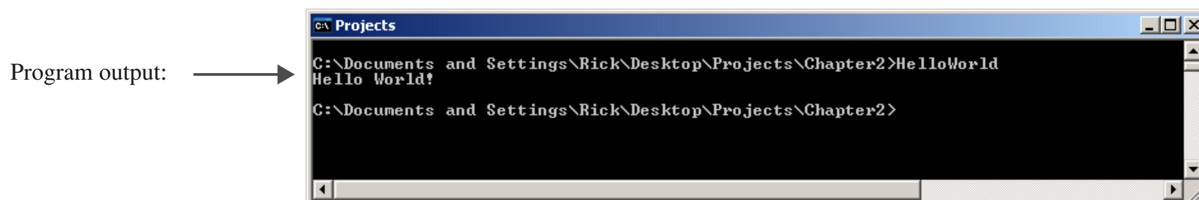


Figure 2-18: Running the HelloWorld Program

Fixing Compilation Errors

No matter how careful you try to be, you're bound to make a mistake or two (or more) when writing programs. Most of these mistakes will be simple typos like forgetting to terminate a statement with a semicolon. When you compile a program that contains a compiler error, you will see something similar to the output shown in Figure 2-19.

When the compiler encounters a problem it will output one or more warning or error messages. Warnings are usually non-fatal in that your program will still run if the compiler signals only a warning message. Error messages, on the other hand, must be addressed before your program will compile completely.

The error message will contain the name of the source file, the line number and character position of the problem, and the compiler error code. The C# compiler error codes can be found on the Microsoft C# language reference site, but searching for them on Microsoft's website does you little good. The best way to find detailed information about a particular C# compiler error is to enter the following search query into Google: "C# compiler error CSNNNN",

 © 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter2>csc HelloWorld.cs
Microsoft (R) Visual C# 2008 Compiler version 3.5.21022.8
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

HelloWorld.cs(6,39): error CS1002: ; expected

C:\Documents and Settings\Rick\Desktop\Projects\Chapter2>
```

Figure 2-19: Compiler Output Showing Compiler Error on Line 6 at Position 39

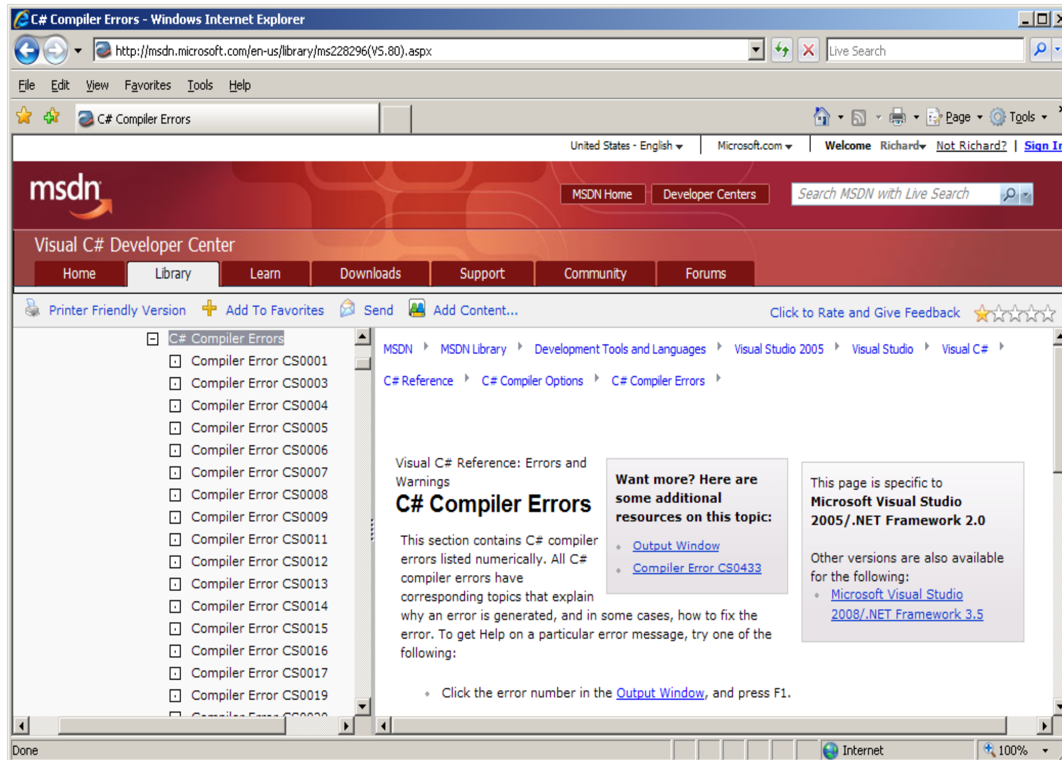


Figure 2-20: C# Language Compiler Errors

where “NNNN” is the compiler error number. The first result from this query will usually lead straight to the Microsoft C# compiler error page for that compiler error number. As you can see from Figure 2-20, the C# compiler’s error message information is located in the Visual C# language reference area. The detailed information for compiler error number CS1002 is shown in Figure 2-21.

Fix The First Compiler Error First

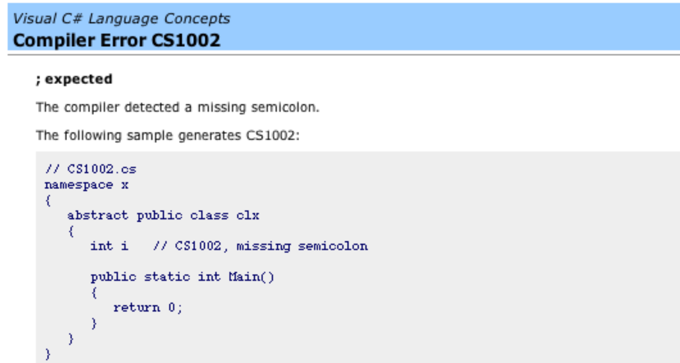
The best advice I can offer when dealing with compiler errors is to always **fix the first compiler error first**. The reason for this is that some compiler errors trigger other errors. Fixing the first error generally eliminates many other errors on the list.

Quick Review

All you need to create robust Microsoft C# applications is a good text editor and the free Microsoft C# command-line compiler that’s included with the .NET Framework Redistributable Package.

You must configure your development environment before using the C# command-line compiler. This includes creating or editing one or more operating system environment variables. An environment variable is a named location

© 2008 Rick Miller — All Rights Reserved


Visual C# Language Concepts
Compiler Error CS1002

; expected

The compiler detected a missing semicolon.

The following sample generates CS1002:

```
// CS1002.cs
namespace x
{
   abstract public class clx
   {
      int i   // CS1002, missing semicolon

      public static int Main()
      {
         return 0;
      }
   }
}
```

Figure 2-21: C# Compiler Error CS1002 “; expected”

in memory used by Microsoft Windows to store data about the operating system environment. There are generally two types of environment variables: *system* and *user*. System environment variables store data that pertains to and affects the operating system environment for all users; user environment variables store data that pertains to and affects the operating system environment for a particular user.

Environment variable values can be accessed by enclosing the variable name in ‘%’ characters.

The operating system uses the Path environment variable to help it find executable files. You must create or edit the Path environment variable to include the full path to the C# compiler (csc.exe).

It’s helpful to create a project folder and a shortcut to the command console on your desktop. Set the command console shortcut’s **Start in** property so it will automatically open in your designated project folder. Increase the command console shortcut’s screen buffer **height** and **width** properties to see more information in the console window.

It’s also a good idea to set your folder options to display file type extensions. This will prevent headaches associated with accidently saving source files with a “.txt” extension.

To create a C# program with the command-line compiler, you must create the source file, compile the source file with the `csc` command-line compiler tool, and then execute the program by typing its name at the command prompt and pressing the Return or Enter key.

You’re bound to get a few compiler errors when you start writing your own programs. Go to Microsoft’s website to look up the error code. Remember to always **fix the first compiler error first**!

Creating Projects With Microsoft Visual C# Express

Microsoft Visual C# Express Edition is an IDE that combines text editing, debugging, project management, and a host of other features. Visual C# Express is a trimmed-down, lightweight version of Microsoft’s flagship development environment Visual Studio. With Visual C# Express, you can create C# console and Windows forms applications. Visual C# Express also comes bundled with Microsoft SQL Server Compact Edition. This allows you to create applications that access and store data to a relational database.

The convenience and power of Microsoft’s Visual Studio products come at a price. Although conceptually they are “easy” to learn, in reality, their multitude of features do present a significant learning curve to those who are absolutely new to programming. The benefit Visual Studio brings to the programmer is its seamless integration of Microsoft’s powerful arsenal of developer tools. Your productivity will exponentially increase when you do make the move from command-line tools to the Visual Studio environment — that is, after you’ve figured out how to properly use the tool.

Download and Install Visual C# Express

I’ll make the assumption in this section that you have a high-speed internet connection. If not, I recommend you order the Visual C# Express DVD from Microsoft.

© 2008 Rick Miller — All Rights Reserved

The download and installation of Visual C# Express is straightforward. Open a web browser and go to MSDN [www.msdn.com]. Under the *Developer Tools and Languages* heading click on Visual C#. This will take you to the Visual C# Developer Center home page. Under the *Get Visual Studio* heading click the Free Download: Visual C# 2008 Express link. Click the Download Now! link and select Visual C# 2008 Express Edition.

When you start the install, you'll eventually be presented with the installation screen shown in Figure 2-22.

Figure 2-22: Visual C# Express Installation Window

Note that the installation also includes Microsoft SQL Server Compact 3.5 and its associated design tools. After installation is complete you will need to restart your computer. Having done this, you should be able to start Visual C# by selecting Start->All Programs->Microsoft Visual C# 2008 Express Edition. You will get a screen that looks similar to Figure 2-23.

Quick Tour Of Visual C# Express

The best way to get a feel for how Visual C# Express works is to create the HelloWorld project. If you haven't already done so, launch Visual Studio Express.

Select Project Type

Start by selecting File->New Project from the main menu. This opens the New Project dialog window shown in Figure 2-24. Click the OK button to create a project. Your Visual C# Express window will now look similar to Figure 2-25. Notice in Figure 2-25 that Visual C# Express automatically generated a lot of source code when it created the new project. But, if you look closely, it added a few more lines of code than what I provided in Example 2.1, and also left one out. Additionally, the name of the class is not HelloWorld. It's "Program"! Also, it did not add the keyword `public` to the beginning of the class declaration, and it used a different version of the `Main()` method. There is also a critical piece of code missing, namely, the `Console.WriteLine("Hello World!");` statement that goes in the `Main()` method. Let's make a few changes to the automatically generated code and then run the project.

First, in the Solution Explorer -HelloWorld pane, find the Program.cs file and right-click it to rename it to "HelloWord.cs". When you do this all references to Program will be automatically updated to HelloWorld. Next, in front of the words `class` and `static` add the keyword `public`. Finally, add the statement `Console.WriteLine("Hello World!");` between the opening and closing brackets of the `Main()` method.

Notice that as you type, Visual C# Express tries to lend a hand with its IntelliSense technology, as is shown in Figure 2-26. Referring to Figure 2-26, IntelliSense is offering a list of the Console object's available public methods

 © 2008 Rick Miller — All Rights Reserved

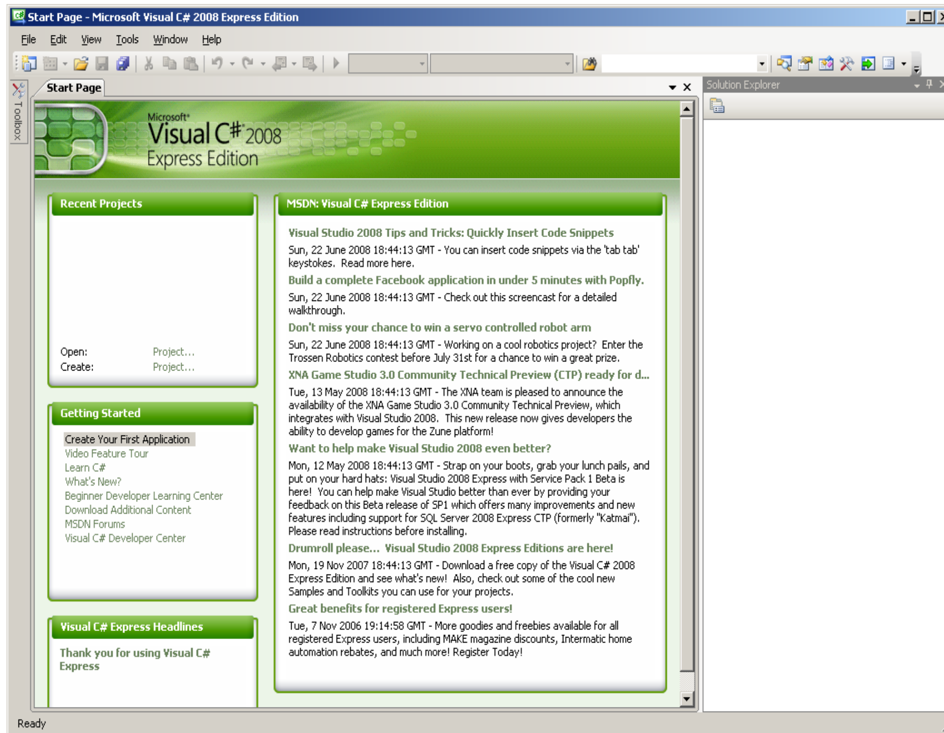


Figure 2-23: Visual C# Express Initial Start-Up Screen

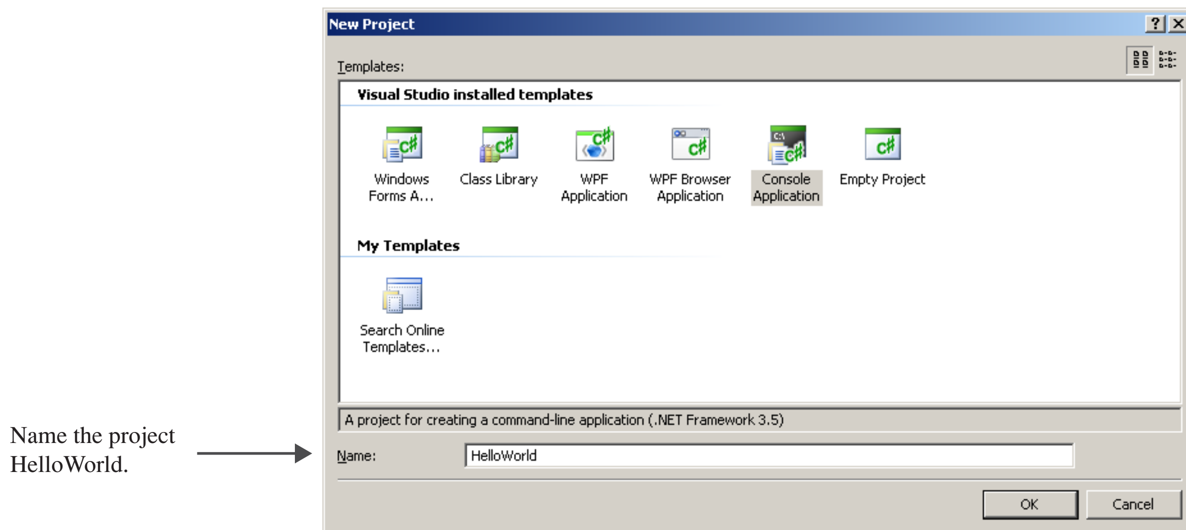


Figure 2-24: New Project Dialog Showing Console Application Selected

and properties. IntelliSense can potentially save you a lot of time looking up .NET Framework class information, but it won't do all the work for you.

When you have finished making the changes to the automatically generated code, the project should look similar to Figure 2-27.

 © 2008 Rick Miller — All Rights Reserved

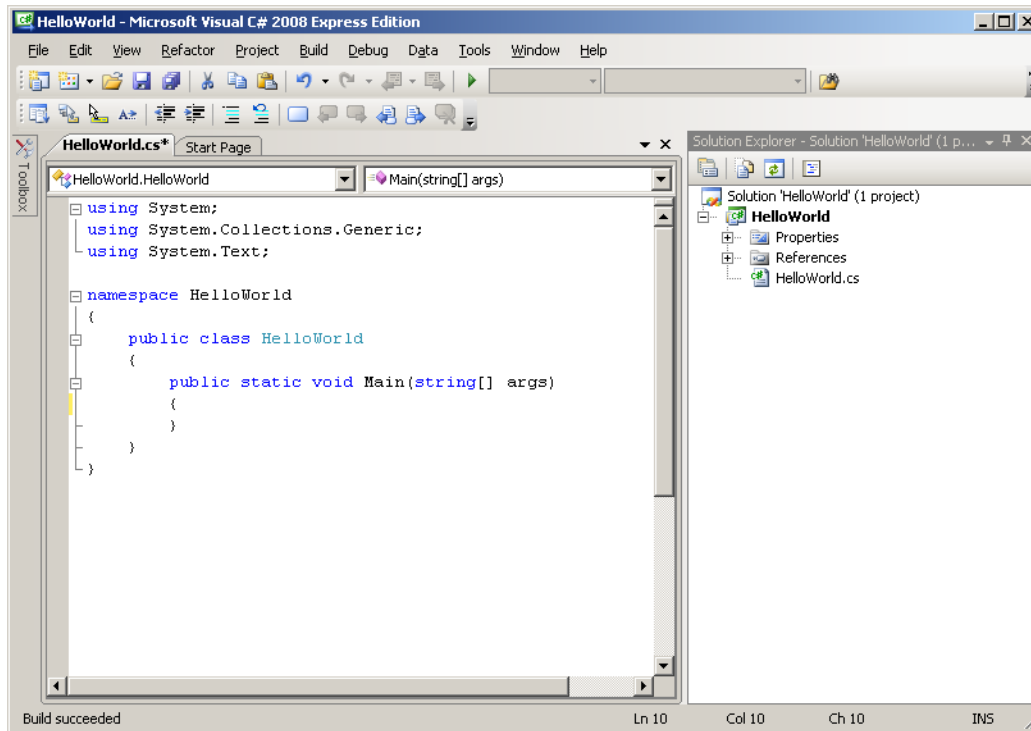


Figure 2-25: HelloWorld Project View

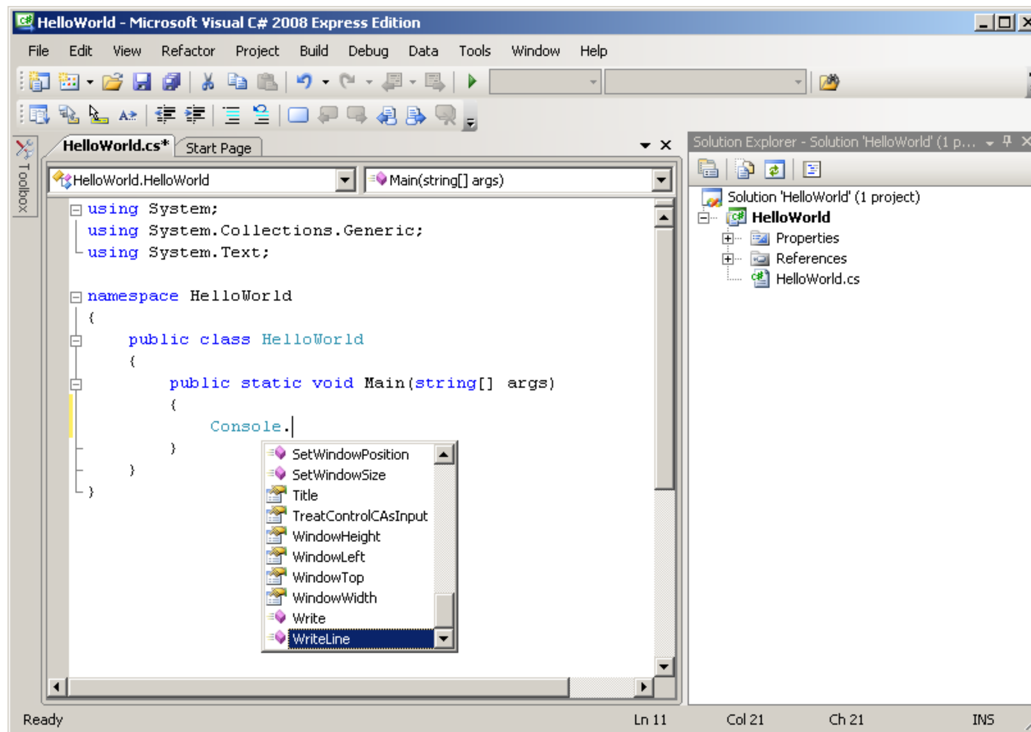


Figure 2-26: IntelliSense Pop-Up Window Showing Available Console Object Methods and Properties

© 2008 Rick Miller — All Rights Reserved

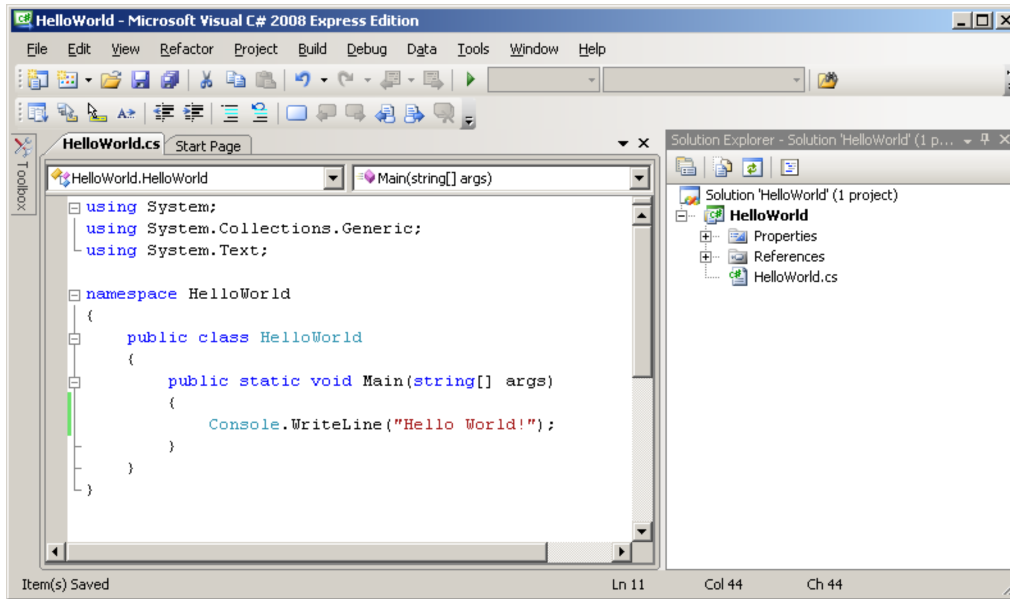


Figure 2-27: Updated HelloWorld Visual C# Project

Saving The Project

Next, save the HelloWorld project by selecting File->Save All from the main menu to open the Save Project dialog. You can create a new folder in your Projects folder and save it there.

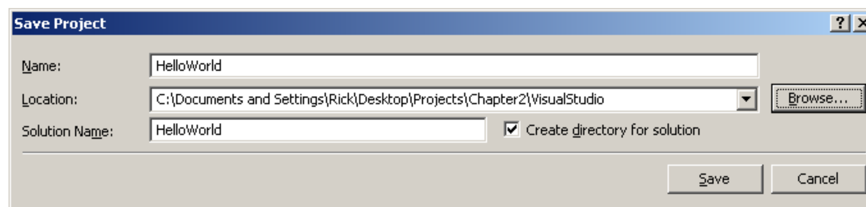


Figure 2-28: Saving the HelloWorld Project

Build The Project

Before you can run the project you must build it. Building the project compiles the source files and generates an executable file (i.e., a .exe file). Select Build->Build Solution from the main menu or press the F6 key as is shown in Figure 2-29.

Locating The Project Executable File

To run the HelloWorld project, navigate to the folder in which Visual C# saved the executable file. You could run the project directly from Visual C# Express. But, because HelloWorld simply prints a short message to the screen and then immediately exits, you'll have to be quick to catch the program's output. This type of short console application is best run from the command line.

Find the HelloWorld.exe file by opening a command console window, changing to the directory in which you saved the project, and then executing the `tree /f` command at the command prompt. This will give you a directory and file listing similar to that shown in Figure 2-30.

Notice that Visual C# Express automatically creates many subdirectories and files in the project's directory. Some of this data is used to maintain project state information, like the number and types of files it contains, along with debugging information.

 © 2008 Rick Miller — All Rights Reserved

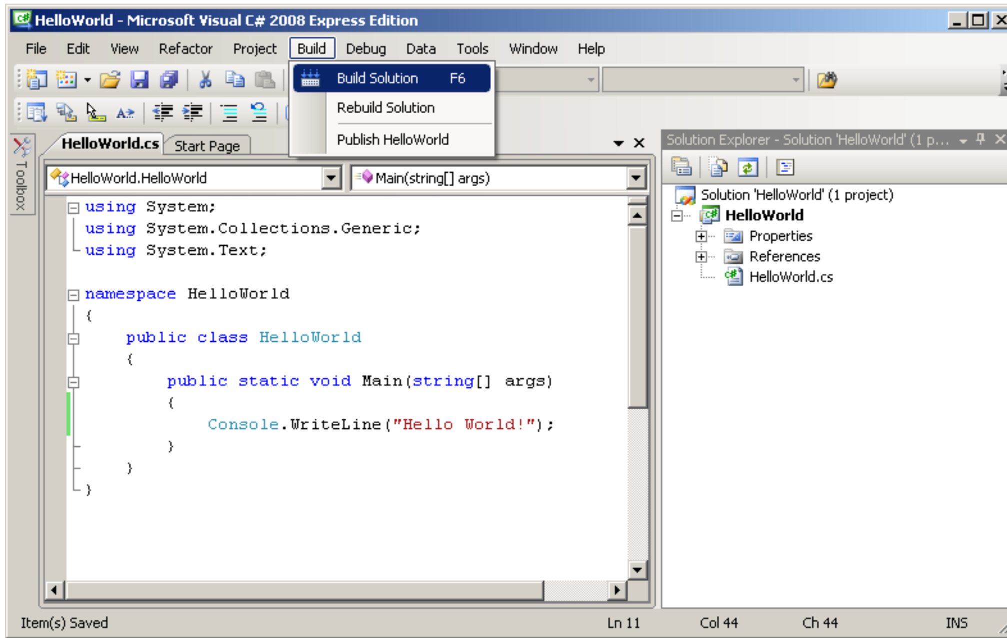


Figure 2-29: Building HelloWorld Project

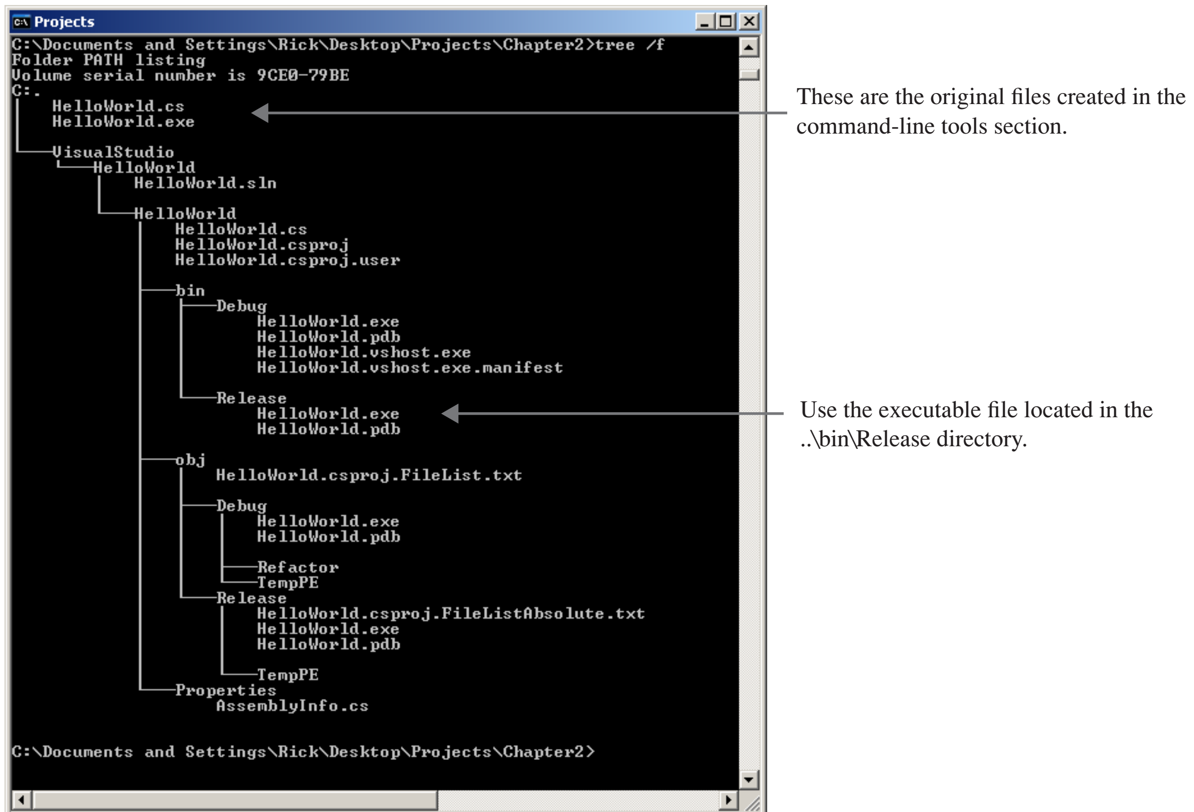


Figure 2-30: Results of Running the `tree /f` Command from the Command Prompt

© 2008 Rick Miller — All Rights Reserved

As you can see from figures 2-28 and 2-30, I have saved my version of the Visual C# Express HelloWorld project in a directory named "VisualStudio" located in the Chapter2 directory of the Projects directory.

Note to Students: One good reason you need to know where to find the executable file is so you can turn in executables from programming assignments to your instructor!

Execute The Project

Execute the HelloWorld program generated by Visual C# Express by opening the command console and navigating to the `..bin\Release` directory using the `cd` command. Enter "HelloWorld" at the command prompt and press Return or Enter. If you want to avoid typing long path names at the command prompt, open the Projects folder and navigate to the `..\Chapter2\VisualStudio\HelloWorld\HelloWorld\bin\Release` folder using Windows Explorer. When you get there, copy the path from the Address bar and paste it into the console window after the `cd` command by right-clicking the console window and clicking paste. (*In this case, using the keyboard shortcut ctrl-v key combination will only paste in the characters "^V", which is not going to help you much.*)

Note: Programs, such as a console application that displays a text menu or a Windows Forms GUI application, that do not exit immediately can be executed successfully from the Visual C# environment.

Where To Go For More Information About Visual C# Express

For more information about Visual C# Express visit the MSDN website or refer to the documentation that came with the application.

Quick Review

Visual C# Express Edition is a lightweight version of Microsoft's flagship IDE Visual Studio. Visual C# Express comes bundled with SQL Server Compact Edition, which allows you to create relational database applications.

Visual C# Express will automatically generate a lot of source code for your project. However, it will not generate all the code your application needs. It's imperative you know how to modify the code that Visual C# generates in order to take control of your projects.

When you save your Visual C# Express project, it generates many different files and subdirectories. You can find the project's executable file in the `..\bin\Release` directory.

Summary

All you need to create robust Microsoft C# applications is a good text editor and the free Microsoft C# command-line compiler that's included with the .NET Framework Redistributable Package.

Before using the C# command-line compiler you must configure your development environment. This includes creating or editing one or more operating system environment variables. An environment variable is a named location in memory used by Microsoft Windows to store data about the operating system environment. There are generally two types of environment variables: system and user. System environment variables store data that pertains to and affects the operating system environment for all users; user environment variables store data that pertains to and affects the operating system environment for a particular user.

Environment variable values can be accessed by enclosing the variable name in '%' characters.

The Path environment variable is used by the operating system to help it locate executable files. You must create or edit the Path environment variable to include the full path to the C# compiler. (csc.exe)

It's helpful to create a project folder and a shortcut to the command console on your desktop. Set the command console shortcut's **Start in** property so it will automatically open in your designated project folder. Increase the command console shortcut's screen buffer **height** and **width** properties to see more information in the console window.

It's also a good idea to set your folder options to display file type extensions. This will prevent headaches associated with accidently saving source files with a ".txt" extension.

 © 2008 Rick Miller — All Rights Reserved

To create a C# program with the command-line compiler you must create the source file, compile the source file with the `csc` command-line compiler tool, and then execute the program by typing its name at the command prompt and pressing the Return or Enter key.

You're bound to get a few compiler errors when you start writing your own programs. Go to Microsoft's website to look up the error code and always remember to **fix the first compiler error first**!

Visual C# Express Edition is a lightweight version of Microsoft's flagship integrated development environment (IDE) Visual Studio. Visual C# Express comes bundled with SQL Server Compact Edition which allows you to create relational database applications.

Visual C# Express automatically generates a lot of source code for your project. However, it will not generate all the code your application needs. It's imperative you know how to modify the code that Visual C# generates in order to take control of your projects.

When you save your Visual C# Express project it generates many different files and sub directories. You can find the project's executable file in the `..\bin\Release` directory.

Skill-Building Exercises

1. **Creating and Using Environment Variables:** Create an environment variable named "PROJECTS_HOME". For its value use the path to your projects folder.

2. **Setting Up Your Development Environment**: Set up your development environment following the steps outlined in this chapter. Test your development environment by compiling and running the program given in Example 2.1.

3. **Web Research**: Visit the MSDN website and familiarize yourself with the information it contains. Locate the C# compiler errors page and bookmark the page in your web browser.

4. **.NET Software Development Kit (SDK)**: Download and install the .NET SDK. List and provide a brief description of the purpose of each component.

Suggested Projects

1. **Alternative .NET Development Environments** - If you are interested in doing C#.NET development on alternative computing platforms, the Mono development environment may help. Visit the Mono Project website [www.mono-project.com] to learn more.

Self-Test Questions

1. What two things, at minimum, do you need to do C#.NET development?

2. What is an operating system environment variable?

3. What is the difference between a user vs. a system environment variable?

4. How do you create an environment variable in Microsoft Windows XP?

5. What characters must you use before and after an environment variable to get its value?

 © 2008 Rick Miller — All Rights Reserved

6. What is the purpose of the Path environment variable?

7. What should you do if you get more than one compiler error?

8. What's the advantage of using an IDE like Visual C# Express?

9. What are the general steps required to create, compile, and execute a C# program?

10. What's the recommended way to run a C# program that runs briefly and exits immediately after executing?

References

Microsoft Developer's Network website, [www.msdn.com]

Mono Project website, [http://www.mono-project.com/Main_Page]

Notes

© 2008 Rick Miller — All Rights Reserved

Chapter 3

Pentax 67 / 50mm Lens / Kodak Tri-X Professional

My Backyard

Project Walkthrough

Learning Objectives

- *Apply the project-approach strategy to implement a C# programming assignment*
- *Apply the development cycle to implement a C# programming assignment*
- *State the actions performed by the development roles of analyst, designer, and programmer*
- *Translate a project specification into a software design that can be implemented in C#*
- *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 csc command-line tool to compile C# source files*
- *Execute C# programs*
- *State the importance of compiling and testing early in the development process*

© 2008 Rick Miller — All Rights Reserved

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 procedure based. I do this because I find that trying simultaneously to teach problem-solving skills, the C# 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, C# forces the issue by requiring that all methods belong to a class. I mitigate this by presenting a solution that results in one user-defined class. As you pursue your studies of C# 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. 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. *This results in a clear problem definition and a list of required project features.*
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. *This results in a high-level solution statement that can be translated into an application design.*
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. *This results in a notional understanding of the language features required to effect a good design and solve the problem.*
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. *This results in a plan of attack!*

Table 3-1: Project Approach Strategy

© 2008 Rick Miller — All Rights Reserved

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 then 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 spiral or *iterative* fashion as depicted in Figure 3-1. By iterative, 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, development progresses until you converge on the final solution.

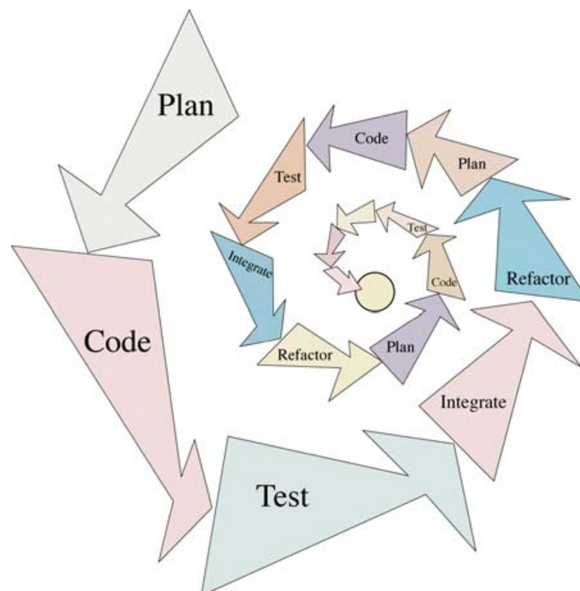


Figure 3-1: Tight-Spiral Development Cycle Deployment

© 2008 Rick Miller — All Rights Reserved

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.

ITP 136 - Introduction To C# Programming
Project 1
Robot Rat

```
    Objectives:
    Demonstrate your ability to utilize the following language features:
       Value types and reference types
       Two-dimensional arrays
       Program flow-control structures
       Class methods
       Class attributes
       Local method variables
       Constructor methods
       Console input and output

    Task:
    You are in command of a robot rat! Write a C# console application that will
allow you to control the rat's movements around a 20 x 20 grid floor.
    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.
    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.

    Hints:
    - The robot rat can move in four directions: north, south, east, and west.
Implement diagonal movement if you desire.
    - Implement the floor as a two-dimensional array of boolean objects.
    - C# provides the System.Console.ReadLine() and WriteLine() methods that make
it easy to read text from, and and write text to, the console.

    At minimum, provide a text-based command menu with the following or similar
command choices:

                    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

© 2008 Rick Miller — All Rights Reserved

ITP 136 - Introduction To C# Programming
Project 1
Robot Rat

When menu choice 6 is selected to print the floor, the result might look some thing like this, assuming you chose '-' to represent a marked area of the floor and '0' to represent an unmarked area. You may use other pattern characters if desired.
.

```
-----000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
```

In this example, the robot rat moved from the upper left-hand corner of the floor five spaces to the east with the pen down.

Table 3-3: Project Specification

Analyzing The Project Specification

Now is a good time to step through the project-approach strategy and analyze the Robot Rat project using each strategy area as a guide, starting with the project's requirements.

Application Requirements Strategy Area

The Robot Rat project seems clear enough, but it omits a few details. It begins with a set of formally stated project objectives. It then states the task you are to execute: namely, to write a program that lets you control a robot rat. But what, exactly, is a robot rat? That's a fair question whose answer requires a bit of abstract thinking. To clarify your understanding of the project's requirements, you decide to ask me a few questions. Your first question is, "Does the robot rat exist?"

If I answered the question by saying, "Well, obviously, the robot rat does not really exist!", I would be insulting you. Why? Because if you are wondering just what a robot rat is, then you are having difficulty abstracting the concept of a robot rat. I would be doing you a better service by saying, "The robot rat exists, but only as a collection of attributes that provide a limited description of the robot rat." I would also add that by writing a program to control the robot rat's movements around the floor, you are actually *modeling* the concept of a robot rat. And since a model of something usually leaves out some level of detail or contains some simplifying assumptions, I will also tell you that the robot rat does not have legs, fur, or a cute little nose.

© 2008 Rick Miller — All Rights Reserved

Another valid requirements question might focus on exactly what is meant by the term *C# console application*. That too is a good question. A C# console application is a program that interacts with the command console using the System.Console.WriteLine() and ReadLine() methods, and optionally can utilize command-line parameters. A C# console program does not usually have a graphical user interface (GUI), although nothing stops you from writing one that does.

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 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 all input as a text string. If you need to convert a text string into another data type, you can use the methods provided by the System.Convert class. 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 that for the purpose of this project the user is perfect. Yet note for the record that this is absolutely not the case in the real world!

To summarize the requirements thus far:

- Write a program that models the concept of a robot rat and its movement upon a floor.
- Think of the robot rat as an abstraction represented by a collection of attributes. (*I discuss these attributes in greater detail in the problem domain section that follows.*)
- Represent the floor in the program as a two-dimensional array of boolean objects.
- 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 C# console application and 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 help jump-start your creativity is to read through the project specification looking for relevant nouns and verbs or verb phrases. A first pass at this activity yields two lists. The list of nouns suggests possible application objects, data types, and object attributes. Nouns also suggest possible names for class and instance fields (*variables and constants*) and method variables. The list of verbs suggests 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 floor pen pen position (up, down) mark program pattern direction (north, south, east, west) menu	move set pen up set pen down mark turn right turn left print floor exit

Table 3-4: Robot Rat Nouns and Verbs

 © 2008 Rick Miller — All Rights Reserved

This list of nouns and verbs is a good starting point. Now that you have it, what should you do with it? Good question. As I mentioned previously, 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 have a direct relationship to a particular application element. Some nouns 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. Figure 3-2 has one for your consideration.

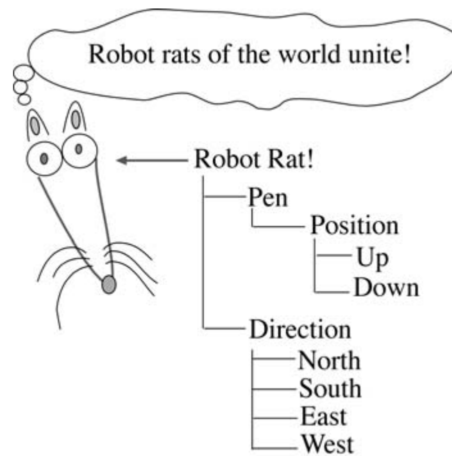


Figure 3-2: Robot Rat Viewed as a Collection of Attributes

Referring to Figure 3-2 — 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 as illustrated in Figure 3-3.

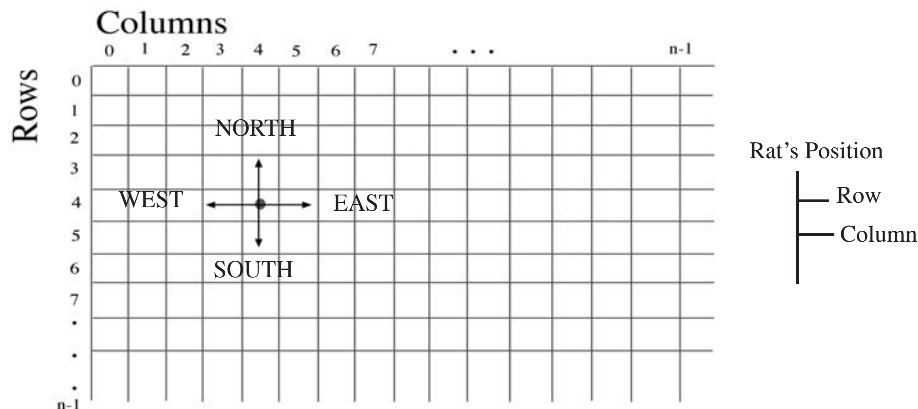


Figure 3-3: Robot Rat Floor Sketch

 © 2008 Rick Miller — All Rights Reserved

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 moves 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 move, its current position on the floor must be determined. Upon completion of each robot rat movement, 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.

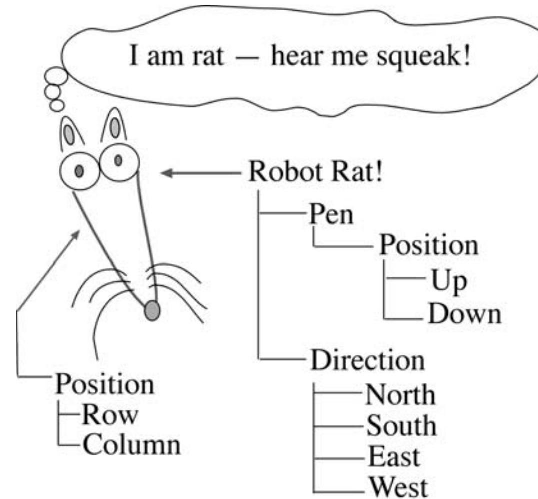


Figure 3-4: Complete Robot Rat Attributes

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 you must understand 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 C# language, learning how to create C# projects, learning an integrated development environment (IDE), 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
	C# applications	How do you write a C# 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 C# application?
	Classes	How do you declare and implement a class? What's the structure of a class. How do you create and compile a C# source file?

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

© 2008 Rick Miller — All Rights Reserved

Check-Off	Feature	Considerations
	Value data types	What is a value 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 value 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 *new* operator? What pre-existing C# classes can be used to help create the program?
	Arrays	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 one-dimensional array and a two-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?
	Properties	What is a property? How do you implement a property? What's the difference between a read-only property and a read-write property?
	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 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 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 to 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.

© 2008 Rick Miller — All Rights Reserved

Design Strategy Area

You must derive a plan of attack before you can solve the robot rat problem! Your 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 both 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. On the other hand, complex projects 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 your project's design will look similar to Figure 3-5.

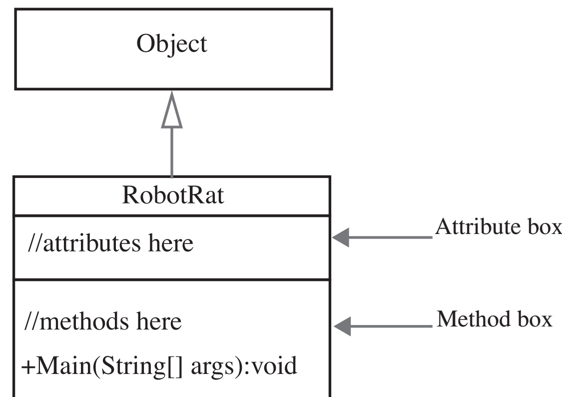


Figure 3-5: RobotRat UML Class Diagram

As Figure 3-5 illustrates, the RobotRat class extends (*inherits*) the functionality provided by the System.Object class. This is indicated by the hollow-pointed arrow pointing from RobotRat to Object. In C#, 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 (*properties and 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 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 easily 90% percent of the battle!

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. C# forces the issue somewhat by requiring that all methods and attributes belong to a class.

There will be many unknowns when you start your design. 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.

 © 2008 Rick Miller — All Rights Reserved

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 you have tested the menu-command processing framework, you must implement each menu item's functionality. This means that you must implement and test the stub methods created in the previous step. 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 repeatedly 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. That is, 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 Robot Rat project's development cycle.

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 another that notes you design decisions 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 (APIs). (*i.e., .NET Framework classes 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 C# application class	The class name will be RobotRat. It will contain a "public static void Main(String[] args){ } method.
	Constructor method	Write a 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 on to the code phase.

© 2008 Rick Miller — All Rights Reserved

Code (First Iteration)

Create the Robot Rat project using your development environment. Create a C# source file named RobotRat.cs and in it create the RobotRat class definition. Add to this class the Main() method and the RobotRat constructor method. When complete, your RobotRat.cs source file should look similar to Example 3.1.

3.1 RobotRat.cs (1st Iteration)

```
1    using System;
2
3    public class RobotRat {
4       public RobotRat(){
5         Console.WriteLine("RobotRat lives!");
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.cs file and running the resulting RobotRat.exe file. If the compilation results in errors, return to the code phase, edit the file to make the necessary correction, and then attempt to compile and test again. Repeat the cycle until you are successful. When you have successfully compiled and tested the first iteration of the RobotRat program, move on to the next step of the development cycle. Figure 3-6 shows the results of running Example 3.1.

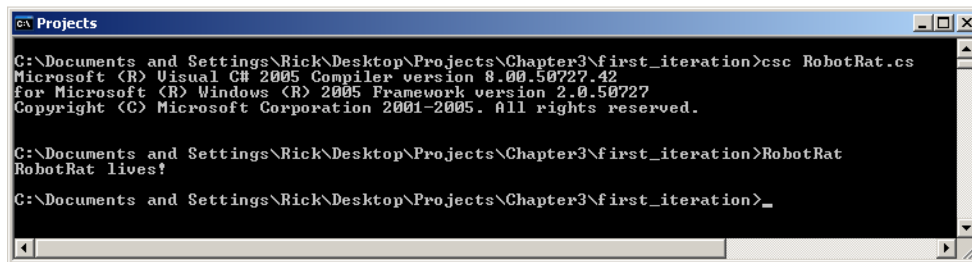


Figure 3-6: Compiling and Testing RobotRat — First Iteration

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 into the project. However, for larger projects, you will code and test a piece of functionality, usually at the class level, 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.

© 2008 Rick Miller — All Rights Reserved

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 Console.WriteLine() 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.cs source file to add a method named PrintMenu(). Example 3.2 shows the RobotRat.cs file with the PrintMenu() method added.

3.2 RobotRat.cs
(2nd Iteration)

```
1    using System;
2
3    public class RobotRat {
4
5       public RobotRat(){
6         PrintMenu();
7       }
8
9       public void PrintMenu(){
10        Console.WriteLine("\n\n");
11        Console.WriteLine("   RobotRat Control Menu");
12        Console.WriteLine();
13        Console.WriteLine("  1. Pen Up");
14        Console.WriteLine("  2. Pen Down");
15        Console.WriteLine("  3. Turn Right");
16        Console.WriteLine("  4. Turn Left");
17        Console.WriteLine("  5. Move Forward");
18        Console.WriteLine("  6. Print Floor");
19        Console.WriteLine("  7. Exit");
20        Console.WriteLine("\n\n");
21      }
22
23   public static void Main(String[] args){
24        RobotRat rr = new RobotRat();
25      }
26   }
```

Referring to Example 3.2 — the PrintMenu() method begins on line 9. Notice how it's using the Console.WriteLine() 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 method has changed. I removed the line printing the "RobotRat Lives!" message and replaced it with a call to the PrintMenu() method on line 6.

When you've finished making the edits to RobotRat.cs, 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.

© 2008 Rick Miller — All Rights Reserved

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter3\second_iteration>RobotRat


  RobotRat Control Menu

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


C:\Documents and Settings\Rick\Desktop\Projects\Chapter3\second_iteration>_
```

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

Integrate/Test (Second Iteration)

Again, there is not much to integrate or 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 that 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	Use the Console.Readline() method to read a string of characters from the console. Once you get the text string from the console, you will only want to use the first letter of the string. Individual string elements can be accessed using array notation. The first character of a string resides at the 0th element. If you use a variable named "input" to hold the string, the first character can be obtained by using "input[0]".
	Execute the user's selected menu command	This will be accomplished with a user-defined method named ProcessMenuChoice(). This method will return void and take no arguments. The body of the ProcessMenuChoice() 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 as follows: SetPenUp(), SetPenDown(), TurnLeft(), TurnRight(), MoveForward(), and PrintFloor(). To repeatedly print the control menu and process user commands you will need to create another method that calls the PrintMenu() and ProcessMenuChoice() methods in a continuous loop until the user exits the application. The name of this method could be called Run().

Table 3-8: Third Iteration Design Considerations

© 2008 Rick Miller — All Rights Reserved

Check-Off	Design Consideration	Design Decision
	Use method stubbing to test the ProcessMenuChoice() method	This means that the user-defined methods mentioned above will be stubbed out. The only functionality they will provide during this iteration will be to print a short test message to the console.
	Exiting the application.	Use a boolean sentinel variable to exit the application. The name of this variable could be called "keep_going", and initially can be set to true. When the user selects the exit menu item, the program will set the value of keep_going to false.
	Program readability	Use character constants to represent the possible menu values and use them in the body of the switch statement. For example, if the user selects the Pen Up menu option he will enter the character '1'. You could use '1' in the switch statement, but, will you remember that '1' represents Pen Up? Instead of embedding the character '1' in the switch statement, you can declare a character constant named "PEN_UP" and set its value to '1'. In the body of the switch statement you can then use the constant and know exactly what menu choice you're talking about. Using constants in this fashion significantly improves program readability.

Table 3-8: Third Iteration Design Considerations

This list of items 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.

Don't try to implement everything on the list completely and then compile. Instead, try to get one or two menu options working with the stubbed methods, then compile and test. When you're happy with the results, add the capability to process another menu option, and so on, and so on, until you've got it licked!

Code (Third Iteration)

Example 3.3 gives the code for the RobotRat.cs file with all the new features implemented. I'll discuss the new code sections following the example.

3.3 RobotRat.cs (3rd Iteration)

```
1    using System;
2
3    public class RobotRat {
4
5       private bool keep_going = true;
6
7       private const char PEN_UP       = '1';
8       private const char PEN_DOWN     = '2';
9       private const char TURN_RIGHT   = '3';
10      private const char TURN_LEFT    = '4';
11      private const char MOVE_FORWARD = '5';
12      private const char PRINT_FLOOR  = '6';
13      private const char EXIT         = '7';
14
15      public RobotRat(){
16        Console.WriteLine("RobotRat Lives!");
17      }
18
19      public void PrintMenu(){
20        Console.WriteLine("\n\n");
21        Console.WriteLine("   RobotRat Control Menu");
22        Console.WriteLine();
23        Console.WriteLine("  1. Pen Up");
24        Console.WriteLine("  2. Pen Down");
25        Console.WriteLine("  3. Turn Right");
26        Console.WriteLine("  4. Turn Left");
27        Console.WriteLine("  5. Move Forward");
28        Console.WriteLine("  6. Print Floor");
29        Console.WriteLine("  7. Exit");
```


© 2008 Rick Miller — All Rights Reserved

```
30        Console.WriteLine("\n\n");
31      }
32
33      public void ProcessMenuChoice(){
34         String input = Console.ReadLine();
35
36         switch(input[0]){
37           case PEN_UP :       SetPenUp();
38                               break;
39           case PEN_DOWN :     SetPenDown();
40                               break;
41           case TURN_RIGHT :   TurnRight();
42                               break;
43           case TURN_LEFT :    TurnLeft();
44                               break;
45           case MOVE_FORWARD : MoveForward();
46                               break;
47           case PRINT_FLOOR :  PrintFloor();
48                               break;
49           case EXIT :         keep_going = false;
50                               break;
51           default :           PrintErrorMessage();
52                               break;
53         }
54      }
55
56
57      public void SetPenUp(){
58        Console.WriteLine("SetPenUp method called.");
59      }
60
61      public void SetPenDown(){
62         Console.WriteLine("SetPenDown method called.");
63      }
64
65      public void TurnRight(){
66        Console.WriteLine("TurnRight method called.");
67      }
68
69      public void TurnLeft(){
70        Console.WriteLine("TurnLeft method called.");
71      }
72
73      public void MoveForward(){
74        Console.WriteLine("MoveForward method called.");
75      }
76
77      public void PrintFloor(){
78        Console.WriteLine("PrintFloor method called.");
79      }
80
81      public void PrintErrorMessage(){
82        Console.WriteLine("Please enter a valid RobotRat control option!");
83      }
84
85      public void Run(){
86        while(keep_going){
87          PrintMenu();
88          ProcessMenuChoice();
89        }
90      }
91
92      public static void Main(String[] args){
93        RobotRat rr = new RobotRat();
94        rr.Run();
95      }
96
97   }
```

Referring to Example 3.3 — notice that the boolean (bool) variable keep_going has been added to the program along with character (char) constants that represent the range of possible menu choices.

© 2008 Rick Miller — All Rights Reserved

The ProcessMenuChoice() method begins on line 33. The first thing it does is declare a local String variable named “input” and assigns to it the character string read from the console with the help of the Console.ReadLine() method. The first character of the input string (input[0]) is then evaluated by the `switch` statement and compared to the values of each of the character constants. Further processing is passed to the appropriate method. For example, in the case of the user selecting menu option ‘1’ for Pen Up, the `switch` statement compares the ‘1’ character to the constant value contained in the PEN_UP constant. This results in a call to the SetPenUp() method. The `break` keyword exists the body of the `switch` statement.

Notice that the `switch` statement contains a `default` case. If the input character fails to match any of the valid menu options, the `default` case executes. This calls the PrintErrorMessage() method, which writes a short message to the console prompting the user to enter a valid menu option.

Let’s now look at how the Run() method works. The Run() method begins on line 85. It contains a `while` loop that is controlled by the value of the keep_going variable. As you can see at the top of the code listing, the value of keep_going is initialized to true. While keep_going is true, the `while` loop will endlessly call the PrintMenu() method followed by the ProcessMenuChoice() method. This will go on until the user enters ‘7’ at the console, which means he wants to exit the program. This causes the value of keep_going to be set to false, at which time the `while` loop in the Run() method exits and the program halts.

Look now at the Main() method on line 92. It creates an instance of RobotRat and calls the Run() method to start the menu display and menu processing cycle.

```
Projects - robotrat
RobotRat Lives!


  RobotRat Control Menu

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


1
SetPenUp method called.


  RobotRat Control Menu

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


2
SetPenDown method called.


  RobotRat Control Menu

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


3
TurnRight method called.


  RobotRat Control Menu

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

Figure 3-8: Testing 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.

Another area where proceeding in small steps is a good idea would be in coding the body of the ProcessMenuChoice() method. You can write the `switch` statement one case at a time. Then, once you gain confidence that your code works as planned, you can code up the remaining cases 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. Note that the addition of one feature affects another program feature. You can adjust the space issue in the next development cycle iteration.

A Bug In The Program

If, while testing, you accidentally pressed the Enter key without typing a menu option, you would have seen the rather disturbing error message shown in Figure 3-9. This message gives you no indication of what went wrong. But if you managed to stay calm and close this window without sending the error report, and looked closely at the command console window in which the Robot Rat program was running, you would see another error message as shown in Figure 3-10. This error message offers a little more information. It says that because you pressed the Enter key with no string to be read in, there was no string from which to access the first character, therefore the attempt to access input[0] resulted in an IndexOutOfRangeException. To fix this problem, you must revisit the ProcessMenuChoice() method and provide some means of either catching the exception and doing something about it, or add some code that prevents the exception from happening in the first place. This latter route is the one I will take in the next example.

 © 2008 Rick Miller — All Rights Reserved

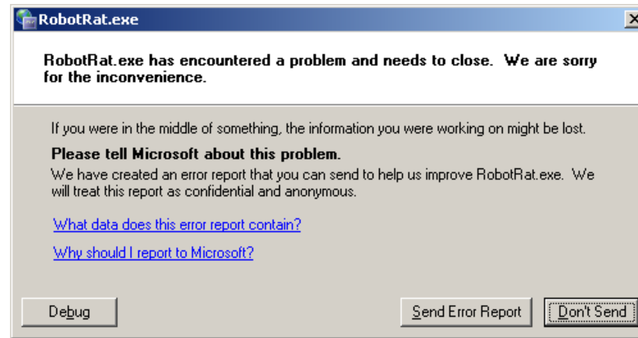


Figure 3-9: A Disturbing Error Message

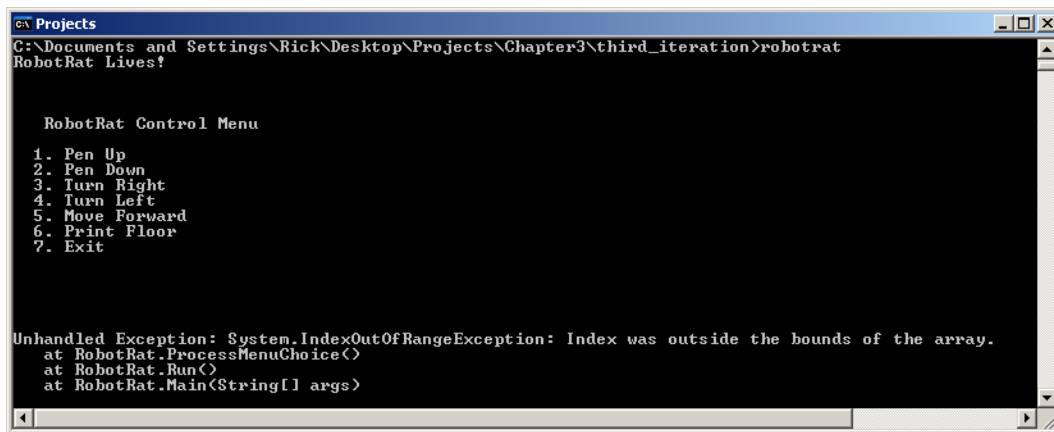


Figure 3-10: Unhandled IndexOutOfRangeException Error Message

Example 3.4 shows the slightly modified ProcessMenuChoice() method with some error prevention code added for good measure.

3.4 Modified ProcessMenuChoice() Method

```
1     public void ProcessMenuChoice(){
2           String input = Console.ReadLine();
3
4           if(input == String.Empty){
5             input = "0";
6           }
7
8           switch(input[0]){
9             case PEN_UP :        SetPenUp();
10                                 break;
11            case PEN_DOWN :      SetPenDown();
12                                 break;
13            case TURN_RIGHT :    TurnRight();
14                                 break;
15            case TURN_LEFT :     TurnLeft();
16                                 break;
17            case MOVE_FORWARD :  MoveForward();
18                                 break;
19            case PRINT_FLOOR :   PrintFloor();
20                                 break;
21            case EXIT :          keep_going = false;
22                                 break;
23            default :            PrintErrorMessage();
24                                 break;
25          }
26     }
```

Referring to Example 3.4 — notice on line 4 that if the input string is empty, I give it the value “0”. This means that if the user fails to enter a valid input string, the `switch` statement’s `default` case will execute and display an error message prompting the user to enter a valid command. Anytime you make a change like this, you must retest your code to make sure everything works fine. This is what is meant by the term *regression testing*.

 © 2008 Rick Miller — All Rights Reserved

Development Cycle: Fourth Iteration

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

Plan (Fourth Iteration)

You now need to both add more attributes to the RobotRat class and begin 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	The direction can be an integer (int) variable or a variable of an enumerated type. It will have four possible states or values: NORTH, SOUTH, EAST, and WEST. You can implement these as class constants or as an enumeration. This will make the source code easier to read and maintain. The robot rat's direction will change when either the TurnLeft() or TurnRight() methods are called. The initial direction upon program startup will be EAST.
	Implement robot rat's pen_position	The pen_position will be an integer variable or a variable of some enumerated type. It will have two valid states or values: UP and DOWN. These can be implemented as class constants or enumerations as well. The robot rat's pen_position will change when either the SetPenUp() or the SetPenDown() methods are called. The initial pen_position value upon program startup will be UP.
	floor	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. The floor array elements upon program startup will be initialized to false.

Table 3-9: Fourth Iteration Design Considerations

This will keep you busy for a while. You may need to spend more time analyzing the issues regarding the setting of the robot rat's direction and its pen_position. It is often helpful to draw state transition diagrams to graphically illustrate object state changes. Figure 3-11 shows the state transition diagram for pen_position.

As Figure 3-11 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 3-12.

As is illustrated in Figure 3-12, 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.

© 2008 Rick Miller — All Rights Reserved

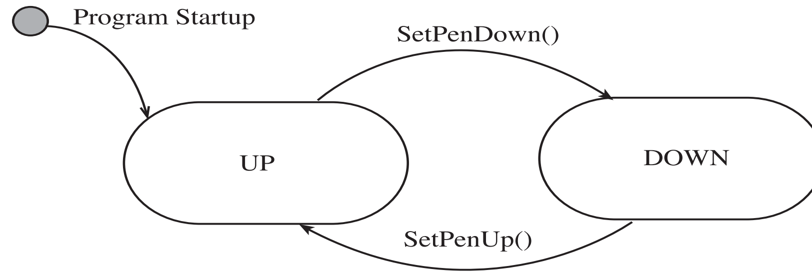


Figure 3-11: pen_position State Transition Diagram

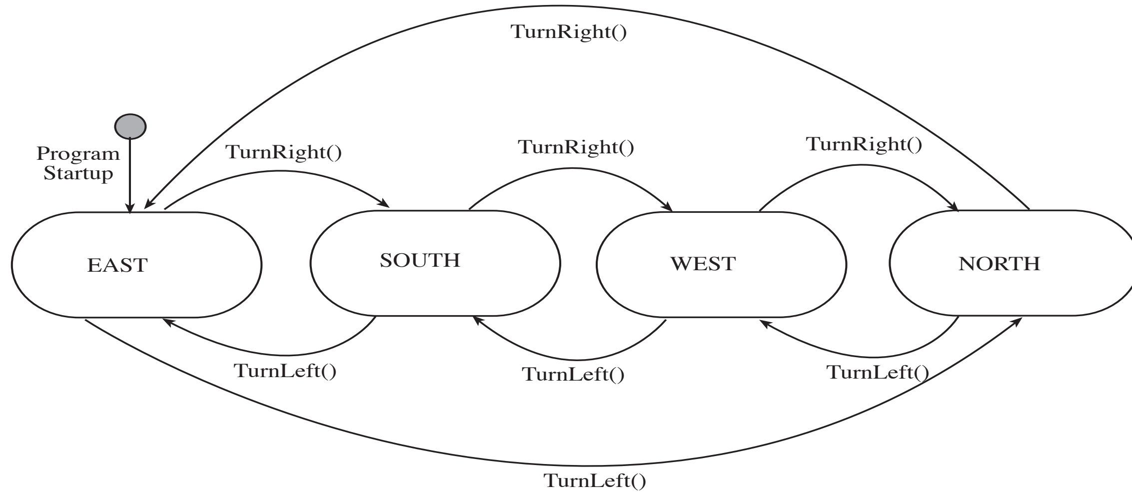


Figure 3-12: State Transition Diagram for the direction Variable

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 3.5 gives a pseudocode description for the TurnLeft() method:

3.5 Pseudocode for 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.
```

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 3.6 gives the pseudocode for the PrintFloor() method.

3.6 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 the 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 robot rat features, you can move on to the code phase of the development cycle.

 © 2008 Rick Miller — All Rights Reserved

Code (Fourth Iteration)

The PrintFloor(), TurnLeft(), TurnRight(), SetPenUp(), and SetPenDown() methods already exist as stub methods. In this phase, you will proceed to add the required code to each of these methods, then compile and test the results. Just like the previous iteration, this code phase comprises multiple code, compile, and test cycles.

To proceed, first add the required variable and enum declarations to the top of the class. Any fields declared here could be initialized either by the constructor method or at the point of declaration. Example 3.7 shows a partial code listing for this section of the RobotRat.cs source file, along with a constructor method that initializes the floor.

3.7 RobotRat.cs
(4th Iteration Partial Listing)

```
1    using System;
2
3    public class RobotRat {
4
5       private bool keep_going = true;
6
7       private const char PEN_UP       = '1';
8       private const char PEN_DOWN     = '2';
9       private const char TURN_RIGHT   = '3';
10      private const char TURN_LEFT    = '4';
11      private const char MOVE_FORWARD = '5';
12      private const char PRINT_FLOOR  = '6';
13      private const char EXIT         = '7';
14
15      private enum PenPositions { UP, DOWN };
16      private enum Directions { NORTH, SOUTH, EAST, WEST };
17
18      private PenPositions pen_position = PenPositions.UP;
19      private Directions   direction    = Directions.EAST;
20
21      private bool[,] floor;
22
23
24      public RobotRat(int rows, int cols){
25        Console.WriteLine("RobotRat Lives!");
26        floor = new bool[rows,cols];
27      }
```

Referring to Example 3.7 — the enumerations *PenPositions* and *Directions* have been declared on lines 15 and 16. Variables of the enum types named *pen_position* and *direction* have been declared and initialized on lines 18 and 19. The floor array is declared on line 21. It is initialized in the constructor method with the help of two constructor parameters named *rows* and *cols*, which represent the number of rows and columns the floor should have when the RobotRat object is created.

Let's look now at the SetPenUp(), SetPenDown(), TurnLeft(), and TurnRight() methods. Example 3.8 shows the source code for the SetPenUp() method.

3.8 SetPenUp() method

```
1    public void SetPenUp(){
2         pen_position = PenPositions.UP;
3         Console.WriteLine("The pen is " + pen_position);
4       }
```

As you can see, it's fairly straightforward. The SetPenUp() method just sets the pen_position attribute to the UP state, and then prints a short message to the console showing the user the current state of the pen_position variable. Example 3.9 gives the code for the SetPenDown() method.

3.9 SetPenDown() method

```
1    public void SetPenDown(){
2          pen_position = PenPositions.DOWN;
3          Console.WriteLine("The pen is " + pen_position);
4       }
```

The SetPenDown() 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 3.10.

3.10 TurnLeft() method

```
1    public void TurnLeft(){
2         switch(direction){
3                case Directions.NORTH : direction = Directions.WEST;
4                                        break;
5                case Directions.WEST  : direction = Directions.SOUTH;
```


© 2008 Rick Miller — All Rights Reserved

```
6                                       break;
7                case Directions.SOUTH : direction = Directions.EAST;
8                                       break;
9                case Directions.EAST  : direction = Directions.NORTH;
10                                      break;
11             }
12
13        Console.WriteLine("Direction is " + direction);
14      }
```

Notice that in the TurnLeft() method the `switch` statement checks the value of the direction field and 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 3.11.

3.11 PrintFloor() method

```
1     public void PrintFloor(){
2            for(int i = 0; i<floor.GetLength(0); i++){
3              for(int j = 0; j<floor.GetLength(1); j++){
4                if(floor[i,j]){
5                  Console.Write('-');
6                }else{
7                  Console.Write('0');
8                }
9              }
10             Console.WriteLine();
11           }
12       }
```

Test (Fourth Iteration)

There's a lot to test for this iteration. You'll need to test all the methods that were modified. You'll be especially anxious to test the PrintFloor() method since now you'll see the floor pattern print to the console. Figure 3-13 shows the PrintFloor() method being tested. As you can see, it just prints the `'0'` characters to the screen. You might find it helpful to load the floor array with a test pattern to test the `'-'` characters as well.

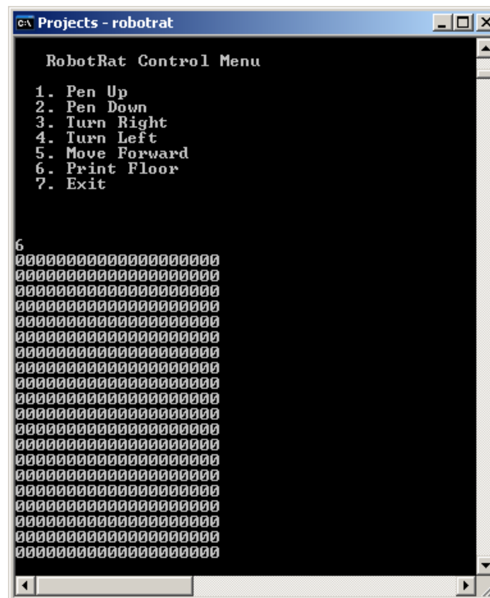


Figure 3-13: Testing the PrintFloor() Method

 © 2008 Rick Miller — All Rights Reserved

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 make it easy to create RobotRat objects with different floor sizes. When you are happy with all the work done in this iteration, it's time to move on to the next development cycle iteration.

Development Cycle: Fifth Iteration

All that remains to code at this point is the MoveForward() 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 its pen position to up or down. The MoveForward() 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 MoveForward() method to properly move the robot rat. For instance, you'll need some way to get the number of spaces from the user to move the robot rat.

Plan (Fifth Iteration)

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

Check-Off	Design Consideration	Design Decision
	Coding the MoveForward() method	Write the code for the MoveForward() method. The MoveForward() method will use the direction and pen_position fields to make move decisions. The MoveForward() method will need a way to get the number of spaces to move from the user. You will have to add the current_row and current_col fields to the RobotRat 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 GetSpacesToMove(). The GetSpacesToMove() method will read a text string from the console and convert it into an integer. The GetSpacesToMove() method will return an integer value and take no parameters.
	Add current_row & current_col fields	The current_row and current_col fields will be declared with the rest of the RobotRat fields and initialized in the constructor or at their point of declaration.

Table 3-10: Fifth Iteration Design Considerations

The first feature from the list to be coded and tested should be the GetSpacesToMove() method. This method will read a string from the console and convert it into an integer value with the help of the System.Convert class.

The MoveForward() method requires 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. You also need to manipulate the floor array element values when moving with the pen in the DOWN position, setting each element to true when the robot rat moves through that position on the floor.

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

© 2008 Rick Miller — All Rights Reserved

Once you get the move mechanics worked out you can write a pseudocode description of the MoveForward() method like the one presented in Example 3.12.

3.12 MoveForward() method pseudocode

```
1    get spaces to move from user
2    if pen_position is UP do the following
3       if direction is NORTH move RobotRat NORTH -- do not mark floor
4       if direction is SOUTH move RobotRat SOUTH -- do not mark floor
5       if direction is EAST move RobotRat EAST -- do not mark floor
6       if direction is WEST move RobotRat WEST -- do not mark floor
7    if pen_position is DOWN
8       if direction is NORTH move RobotRat NORTH -- mark floor
9       if direction is SOUTH move RobotRat SOUTH -- mark floor
10      if direction is EAST move RobotRat EAST -- mark floor
11      if direction is WEST move RobotRat WEST -- mark floor
```

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

Code (Fifth Iteration)

Example 3.13 gives the source code for the GetSpacesToMove() method.

3.13 GetSpacesToMove() method

```
1    public int GetSpacesToMove(){
2       int spaces = 0;
3       String input;
4
5       Console.WriteLine("Please enter number of spaces to move: ");
6       input = Console.ReadLine();
7
8       if(input == String.Empty){
9         spaces = 0;
10      }else{
11         try{
12           spaces = Convert.ToInt32(input);
13
14         }catch(Exception){
15           spaces = 0;
16         }
17      }
18
19      return spaces;
20   }
```

Referring to Example 3.13 — two local variables are declared, one of type int named *spaces*, and the other of type String named *input*. The Console.ReadLine() method on line 6 reads a string from the console after the user has been prompted to enter the number of spaces to move. On line 8, the value of the input variable is compared to String.Empty. If it's empty, spaces is set to 0. If it's not empty, the code tries to convert the string into an integer. Notice that the statement that actually performs the conversion, line 12, is enclosed in a try/catch block. This is necessary because although the input string may not be empty, it may contain a string that does not properly convert into an integer. If an exception is thrown, the spaces variable is set to 0. Finally, the method returns the value of spaces.

The GetSpacesToMove() method can now be used in the MoveForward() method as is shown in Example 3.14.

3.14 MoveForward() method

```
1    public void MoveForward(){
2          int spaces_to_move = GetSpacesToMove();
3
4          switch(pen_position){
5            case PenPositions.UP   :
6                 switch(direction){
7                    case Directions.NORTH :
8                         if((current_row - spaces_to_move) < 0){
9                              current_row = 0;
10                          }else{
11                             current_row = current_row - spaces_to_move;
12                           }
13                           break;
14                    case Directions.SOUTH :
15                          if((current_row + spaces_to_move) > (floor.GetLength(1) - 1)){
16                   current_row = (floor.GetLength(1) - 1);
17                }else{
18             current_row = current_row + spaces_to_move;
19                           }
20                           break;
```

 © 2008 Rick Miller — All Rights Reserved

```
21                    case Directions.EAST :
22                         if((current_col + spaces_to_move) > (floor.GetLength(0) - 1)){
23                             current_col = (floor.GetLength(0) - 1);
24                 }else{
25              current_col = current_col + spaces_to_move;
26                           }
27                           break;
28                    case Directions.WEST :
29                         if((current_col - spaces_to_move) < 0){
30                             current_col = 0;
31                          }else{
32                            current_col = current_col - spaces_to_move;
33                          }
34                         break;
35                    }
36                    break;
37          case PenPositions.DOWN :
38               switch(direction){
39                  case Directions.NORTH :
40                       while((current_row > 0) && (spaces_to_move-- > 0)){
41                          floor[current_row--, current_col] = true;
42                        }
43                       break;
44                   case Directions.SOUTH :
45                        while((current_row < floor.GetLength(0) - 1) && (spaces_to_move-- > 0)){
46                 floor[current_row++, current_col] = true;
47                         }
48                         break;
49                   case Directions.EAST :
50                        while((current_col < floor.GetLength(1) - 1) && (spaces_to_move-- > 0)){
51            floor[current_row, current_col++] = true;
52                         }
53                         break;
54                   case Directions.WEST :
55                        while((current_col > 0) && (spaces_to_move-- > 0)){
56           floor[current_row, current_col--] = true;
57                         }
58                         break;
59                 }
60                 break;
61         }
62       }
```

Referring to Example 3.14 — the MoveForward() method contains nested `switch` statements. The outer `switch` statement evaluates the value of the robot rat's pen_position field. The inner `switch` statements evaluate the value of the robot rat's direction field. Thus, the code performs the appropriate form of movement processing by controlling the value of these two variables by changing the position of the pen (UP or DOWN) and the robot rat's direction (NORTH, SOUTH, EAST, or WEST).

Test (Fifth Iteration)

This will be the most extensive test session of the RobotRat project yet. You must test the MoveForward() method in all directions and ensure you are properly handling move requests that attempt to go beyond the bounds of the floor array. Figure 3-14 shows a screen shot of the floor after completing a series of moves.

Integrate/Test (Fifth Iteration)

At this point in the development cycle, you will want to test the entire integrated 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, noting the effects of the latest features on existing functionality.

© 2008 Rick Miller — All Rights Reserved

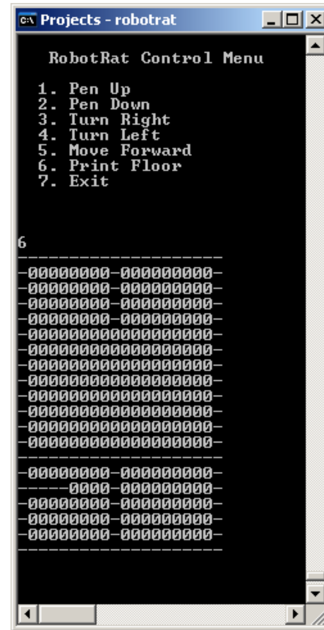


Figure 3-14: Testing Robot Rat Movement in All Directions

Final Considerations

You have now completed the core development efforts, but I would hesitate calling the Robot Rat project finished just yet. After 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 review before submitting your project to your instructor.

Check-Off	Review	What To Check For
	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.
	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 disk, CD-ROM, or USB memory stick (i.e., removable media)	Ensure all the required source and executable files are present. Try running your project from the removable medium to make sure you have included all the required files.

Table 3-11: Final Project Review Checklist

 © 2008 Rick Miller — All Rights Reserved

Complete RobotRat.cs Source Code Listing

3.15 RobotRat.cs (Complete Listing)

```
1    using System;
2
3    /// <summary>
4    /// RobotRat class lets a user control the movements
5    /// of a robot rat around a floor represented by
6    /// a 2-dimensional array.
7    /// </summary>
8    public class RobotRat
9    {
10
11       private bool keep_going = true;
12       private const char PEN_UP = '1';
13       private const char PEN_DOWN = '2';
14       private const char TURN_RIGHT = '3';
15       private const char TURN_LEFT = '4';
16       private const char MOVE_FORWARD = '5';
17       private const char PRINT_FLOOR = '6';
18       private const char EXIT = '7';
19       private enum PenPositions { UP, DOWN };
20       private enum Directions { NORTH, SOUTH, EAST, WEST };
21       private PenPositions pen_position = PenPositions.UP;
22       private Directions direction = Directions.EAST;
23       private bool[,] floor;
24       private int current_row = 0;
25       private int current_col = 0;
26
27       /// <summary>
28       /// Constructor method.
29       /// </summary>
30       /// <param name="rows">Integer value representing number of floor rows</param>
31       /// <param name="cols">Integer value representing number of floor columns</param>
32       public RobotRat(int rows, int cols)
33       {
34           Console.WriteLine("RobotRat Lives!");
35           floor = new bool[rows, cols];
36       }
37
38       /// <summary>
39       /// Prints the menu to the screen.
40       /// </summary>
41       public void PrintMenu()
42       {
43           Console.WriteLine("\n\n");
44           Console.WriteLine("   RobotRat Control Menu");
45           Console.WriteLine();
46           Console.WriteLine("  1. Pen Up");
47           Console.WriteLine("  2. Pen Down");
48           Console.WriteLine("  3. Turn Right");
49           Console.WriteLine("  4. Turn Left");
50           Console.WriteLine("  5. Move Forward");
51           Console.WriteLine("  6. Print Floor");
52           Console.WriteLine("  7. Exit");
53           Console.WriteLine("\n\n");
54       }
55
56       /// <summary>
57       /// Processes user's menu selection.
58       /// </summary>
59       public void ProcessMenuChoice()
60       {
61           String input = Console.ReadLine();
62
63           if (input == String.Empty)
64           {
65               input = "0";
66           }
67
68           switch (input[0])
69           {
70               case PEN_UP: SetPenUp();
71                   break;
72               case PEN_DOWN: SetPenDown();
73                   break;
```

 © 2008 Rick Miller — All Rights Reserved

```
74              case TURN_RIGHT: TurnRight();
75                  break;
76              case TURN_LEFT: TurnLeft();
77                  break;
78              case MOVE_FORWARD: MoveForward();
79                  break;
80              case PRINT_FLOOR: PrintFloor();
81                  break;
82              case EXIT: keep_going = false;
83                  break;
84              default: PrintErrorMessage();
85                  break;
86          }
87      }
88
89      /// <summary>
90      /// Sets the pen to the UP state.
91      /// </summary>
92      public void SetPenUp()
93      {
94          pen_position = PenPositions.UP;
95          Console.WriteLine("The pen is " + pen_position);
96      }
97
98      /// <summary>
99      /// Sets the pen to the DOWN state.
100     /// </summary>
101     public void SetPenDown()
102     {
103         pen_position = PenPositions.DOWN;
104         Console.WriteLine("The pen is " + pen_position);
105     }
106
107
108     /// <summary>
109     /// Turns the robot rat right.
110     /// </summary>
111     public void TurnRight()
112     {
113         switch (direction)
114         {
115             case Directions.NORTH: direction = Directions.EAST;
116                 break;
117             case Directions.EAST: direction = Directions.SOUTH;
118                 break;
119             case Directions.SOUTH: direction = Directions.WEST;
120                 break;
121             case Directions.WEST: direction = Directions.NORTH;
122                 break;
123         }
124
125         Console.WriteLine("Direction is " + direction);
126     }
127
128
129
130     /// <summary>
131     /// Turns the robot rat left.
132     /// </summary>
133     public void TurnLeft()
134     {
135         switch (direction)
136         {
137             case Directions.NORTH: direction = Directions.WEST;
138                 break;
139             case Directions.WEST: direction = Directions.SOUTH;
140                 break;
141             case Directions.SOUTH: direction = Directions.EAST;
142                 break;
143             case Directions.EAST: direction = Directions.NORTH;
144                 break;
145         }
146
147         Console.WriteLine("Direction is " + direction);
148     }
149
150
151     /// <summary>
152     /// Prints the floor pattern to the console.
153     /// </summary>
154     public void PrintFloor()
```

 © 2008 Rick Miller — All Rights Reserved

```
155        {
156            for (int i = 0; i < floor.GetLength(0); i++)
157            {
158                for (int j = 0; j < floor.GetLength(1); j++)
159                {
160                    if (floor[i, j])
161                    {
162                        Console.Write('-');
163                    }
164                    else
165                    {
166                        Console.Write('0');
167                    }
168                }
169                Console.WriteLine();
170            }
171        }
172
173        /// <summary>
174        /// Prints an error message. Called if a user enters an invalid
175        /// menu choice.
176        /// </summary>
177        public void PrintErrorMessage()
178        {
179            Console.WriteLine("Please enter a valid RobotRat control option!");
180        }
181
182
183        /// <summary>
184        /// This method continuously displays the menu
185        /// and processes user menu choices.
186        /// </summary>
187        public void Run()
188        {
189            while (keep_going)
190            {
191                PrintMenu();
192                ProcessMenuChoice();
193            }
194        }
195
196        /// <summary>
197        /// Called to move the robot rat forward.
198        /// </summary>
199        public void MoveForward()
200        {
201            int spaces_to_move = GetSpacesToMove();
202
203            switch (pen_position)
204            {
205                case PenPositions.UP: switch (direction)
206                    {
207                        case Directions.NORTH:
208                            if ((current_row - spaces_to_move) < 0)
209                            {
210                                current_row = 0;
211                            }
212                            else
213                            {
214                                current_row = current_row - spaces_to_move;
215                            }
216                            break;
217                        case Directions.SOUTH:
218                            if ((current_row + spaces_to_move) > (floor.GetLength(0) - 1))
219                            {
220                                current_row = (floor.GetLength(1) - 1);
221                            }
222                            else
223                            {
224                                current_row = current_row + spaces_to_move;
225                            }
226                            break;
227                        case Directions.EAST:
228                            if ((current_col + spaces_to_move) > (floor.GetLength(1) - 1))
229                            {
230                                current_col = (floor.GetLength(0) - 1);
231                            }
232                            else
233                            {
234                                current_col = current_col + spaces_to_move;
235                            }
```

 © 2008 Rick Miller — All Rights Reserved

```
236                         break;
237                     case Directions.WEST:
238                         if ((current_col - spaces_to_move) < 0)
239                         {
240                             current_col = 0;
241                         }
242                         else
243                         {
244                             current_col = current_col - spaces_to_move;
245                         }
246                         break;
247                 }
248                 break;
249             case PenPositions.DOWN: switch (direction)
250                 {
251                     case Directions.NORTH:
252                         while ((current_row > 0) && (spaces_to_move-- > 0))
253                         {
254                             floor[current_row--, current_col] = true;
255                         }
256                         break;
257                     case Directions.SOUTH:
258                         while ((current_row < floor.GetLength(0) - 1) && (spaces_to_move-- > 0))
259                         {
260                             floor[current_row++, current_col] = true;
261                         }
262                         break;
263                     case Directions.EAST:
264                         while ((current_col < floor.GetLength(1) - 1) && (spaces_to_move-- > 0))
265                         {
266                             floor[current_row, current_col++] = true;
267                         }
268                         break;
269                     case Directions.WEST:
270                         while ((current_col > 0) && (spaces_to_move-- > 0))
271                         {
272                             floor[current_row, current_col--] = true;
273                         }
274                         break;
275                 }
276                 break;
277 
278 
279         }
280     }
281 
282     /// <summary>
283     ///  Gets the number of spaces to move from the user.
284     /// </summary>
285     /// <returns></returns>
286     public int GetSpacesToMove()
287     {
288         int spaces = 0;
289         String input;
290 
291         Console.WriteLine("Please enter number of spaces to move: ");
292         input = Console.ReadLine();
293 
294         if (input == String.Empty)
295         {
296             spaces = 0;
297         }
298         else
299         {
300             try
301             {
302                 spaces = Convert.ToInt32(input);
303 
304             }
305             catch (Exception)
306             {
307                 spaces = 0;
308             }
309         }
310 
311         return spaces;
312     }
313 
314 
315     /// <summary>
316     /// The RobotRat's Main method.
```

 © 2008 Rick Miller — All Rights Reserved

```
317      /// </summary>
318      /// <param name="args"></param>
319      public static void Main(String[] args)
320      {
321          RobotRat rr = new RobotRat(20, 20);
322          rr.Run();
323      }
324
325  }
```

This listing was automatically formatted with Microsoft C# Express Edition. Comments were added to describe the function of each method. You can automatically generate Extensible Markup Language (XML) documentation for this class file by compiling the RobotRat.cs file with the /doc:[documentation output filename] option. For example, to generate an XML documentation file named robotrat_docs.xml compile the robotrat.cs file by entering the following at the command line:

```
csc robotrat.cs /doc:robotrat_docs.xml
```

Example 3.16 shows the results.

3.16 robotrat_docs.xml

```
1    <?xml version="1.0"?>
2    <doc>
3        <assembly>
4            <name>RobotRat</name>
5        </assembly>
6        <members>
7            <member name="T:RobotRat">
8                <summary>
9                RobotRat class lets a user control the movements
10               of a robot rat around a floor represented by
11               a 2-dimensional array.
12               </summary>
13           </member>
14           <member name="M:RobotRat.#ctor(System.Int32,System.Int32)">
15               <summary>
16               Constructor method.
17               </summary>
18               <param name="rows">Integer value representing number of floor rows</param>
19               <param name="cols">Integer value representing number of floor columns</param>
20           </member>
21           <member name="M:RobotRat.PrintMenu">
22               <summary>
23               Prints the floor pattern to the screen.
24               </summary>
25           </member>
26           <member name="M:RobotRat.ProcessMenuChoice">
27               <summary>
28               Processes user's menu selection.
29               </summary>
30           </member>
31           <member name="M:RobotRat.SetPenUp">
32               <summary>
33               Sets the pen to the UP state.
34               </summary>
35           </member>
36           <member name="M:RobotRat.SetPenDown">
37               <summary>
38               Sets the pen to the DOWN state.
39               </summary>
40           </member>
41           <member name="M:RobotRat.TurnRight">
42               <summary>
43               Turns the robot rat right.
44               </summary>
45           </member>
46           <member name="M:RobotRat.TurnLeft">
47               <summary>
48               Turns the robot rat left.
49               </summary>
50           </member>
51           <member name="M:RobotRat.PrintFloor">
52               <summary>
53               Prints the floor pattern to the console.
54               </summary>
55           </member>
56           <member name="M:RobotRat.PrintErrorMessage">
57               <summary>
```


© 2008 Rick Miller — All Rights Reserved

```
58                  Prints an error message. Called if a user enters an invalid
59                  menu choice.
60                  </summary>
61              </member>
62              <member name="M:RobotRat.Run">
63                  <summary>
64                  This method continuously displays the menu
65                  and processes user menu choices.
66                  </summary>
67              </member>
68              <member name="M:RobotRat.MoveForward">
69                  <summary>
70                  Called to move the robot rat forward.
71                  </summary>
72              </member>
73              <member name="M:RobotRat.GetSpacesToMove">
74                  <summary>
75                   Gets the number of spaces to move from the user.
76                  </summary>
77                  <returns></returns>
78              </member>
79              <member name="M:RobotRat.Main(System.String[])">
80                  <summary>
81                  The RobotRat's Main method.
82                  </summary>
83                  <param name="args"></param>
84              </member>
85          </members>
86      </doc>
```

Now, XML isn't pretty to look at and is not at all easily readable by humans. If you want to create professional grade documentation in HTML or another format, you can use an open-source documentation generator like NDoc [http://ndoc.sourceforge.net] or Doxygen [http://www.stack.nl/~dimitri/doxygen/]. **Note:** At the time of this writing NDoc does not work with .NET Framework version 2.0 or greater. These tools generate documentation from commented source files. Figure 3-15 shows a partial screen capture from the documentation generated from the final version of the RobotRat.cs file using Doxygen on a Macintosh running OSX 10.3.

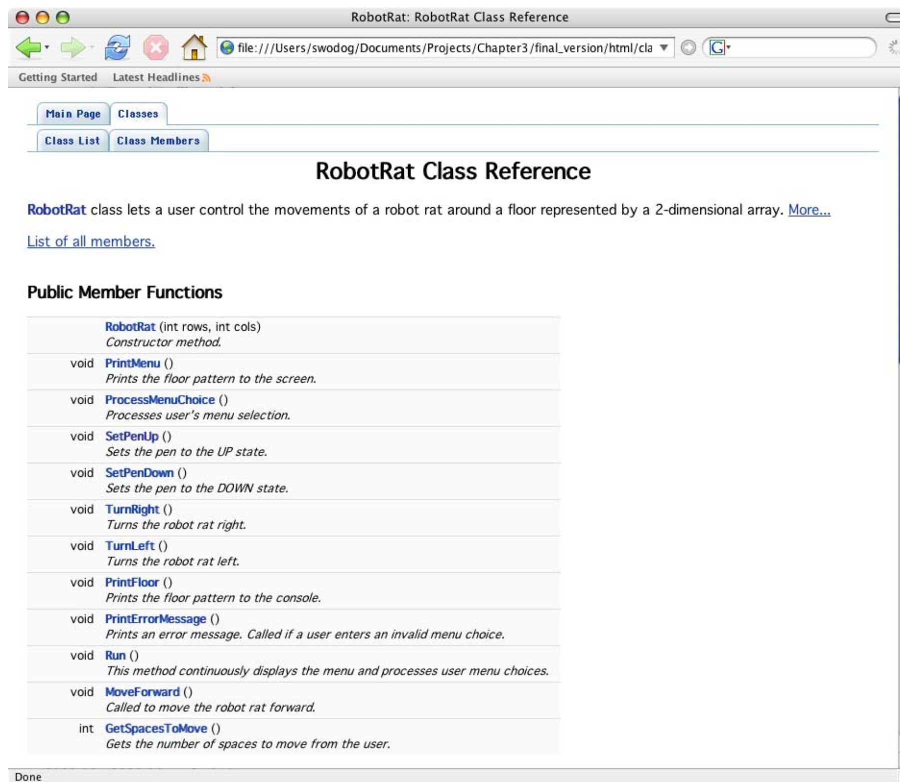


Figure 3-15: Robot Rat HTML Documentation Generated with Doxygen

 © 2008 Rick Miller — All Rights Reserved

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, and 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 floppy disk or another type of removable medium, double-check to ensure you've included all the files necessary to run the project.

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 UML tool. Explore it capabilities. Use it to create class and state transition diagrams.

5. **Documentation Generator:** Download and install one of the documentation generators mentioned in this chapter. Try generating HTML documentation from the final version of the RobotRat.cs file given in Example 3.15.

6. **Microsoft's Documentation Generator:** Microsoft removed documentation generating capabilities from Visual Studio 2005. However, they made their internal documentation generator, Sandcastle, available for public download from their website. Download Sandcastle and try to use it to generate RobotRat class documentation.

Suggested Projects

1. **Project-Approach Strategy and 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.

© 2008 Rick Miller — All Rights Reserved

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. 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

Microsoft Developer Network (MSDN) [www.msdn.com]

Doxygen website [http://www.stack.nl/~dimitri/doxygen/]

NDoc SourceForge project website: [http://ndoc.sourceforge.net]

Notes

© 2008 Rick Miller — All Rights Reserved

Chapter 4

Contax T3 / Kodak Tri-X

Big Snow

Computers, Programs And 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 both 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 Microsoft .NET Common Language Runtime (CLR)*
- *List the similarities between a virtual machine and a real computer*
- *Explain the purpose of Microsoft Intermediate Language (MIL)*
- *Define the concept of an algorithm*

© 2008 Rick Miller — All Rights Reserved

Introduction

Computers, programs, and algorithms are three closely related topics that deserve special attention before you start learning about C# 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 C# programs are transformed into intermediate language and executed by the .NET Common Language Runtime (CLR). Armed with a fundamental understanding of computers, programs, and algorithms, you will be better prepared to understand the concepts of a *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. This is what makes a computer a truly amazing device — it is a changeable machine.

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 C# 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. In this chapter I use the Apple Mac Pro® as an example, but the concepts are the same for any computer or computer system.

Computer System

A typical Apple Mac Pro computer system is pictured in Figure 4-1.

Figure 4-1: Typical Apple Mac Pro Computer System

 © 2008 Rick Miller — All Rights Reserved

Referring to Figure 4-1 — the *computer system* comprises the system unit, monitor, wireless keyboard, mouse, and any other peripheral devices. The computer system also includes any operating system or utility software required to make all the components work together.

The *system unit* houses dual microprocessors, the 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.

Images courtesy Apple Computer, Inc.

Figure 4-2: System Unit Components

The dual microprocessors, or simply *processors*, are connected to the system unit's main logic board with the help of a set of specialized chips referred to as a *chipset*. Different types of microprocessors require different chipsets to help integrate them into the computer system. Electronic pathways called *buses* connect the processor to the various interface components. Other miscellaneous electronic components located on the main logic board control the flow of communication between the processors and the outside world. Figure 4-3 shows a block diagram of a Mac Pro main logic board.

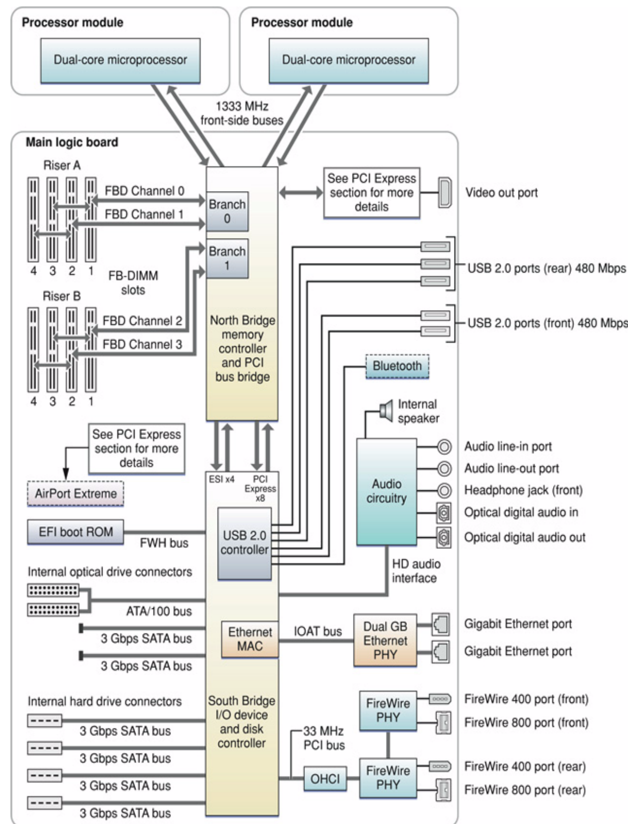


Image courtesy Apple Computer, Inc.

Figure 4-3: Main Logic Board Block Diagram

 © 2008 Rick Miller — All Rights Reserved

Figure 4-3 does a good job of highlighting the number of computer system support components required to help the processors do their 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 high-speed Ethernet ports, keyboard, mouse, speakers, microphones, FireWire devices, and third-party system expansion cards. The heart of the system, however, consists of two Intel Xeon™ 5100 dual-core microprocessors. Let's take a closer look.

Processor

The Intel Xeon 5100 dual-core microprocessor pictured in Figure 4-4 is a 64-bit computer that contains two execution cores in one physical package. Furthermore, the Xeon 5100's design provides two logical processors for each execution core. The logical processors are utilized by Intel's Hyper-Threading Technology (HTT) to increase overall instruction processing throughput in multithreaded software applications.

Figure 4-5 shows a simplified block diagram of the Xeon 5100 dual-core processor architecture. As you can see in Figure 4-5, a computer system containing two Xeon 5100 processors actually has four execution units and eight logical processors. **Note:** The acronym APIC stands for Advanced Programmable Interrupt Controller.

Image courtesy of Intel Corp.

Figure 4-4: Intel Xeon 5100 Dual core Processor

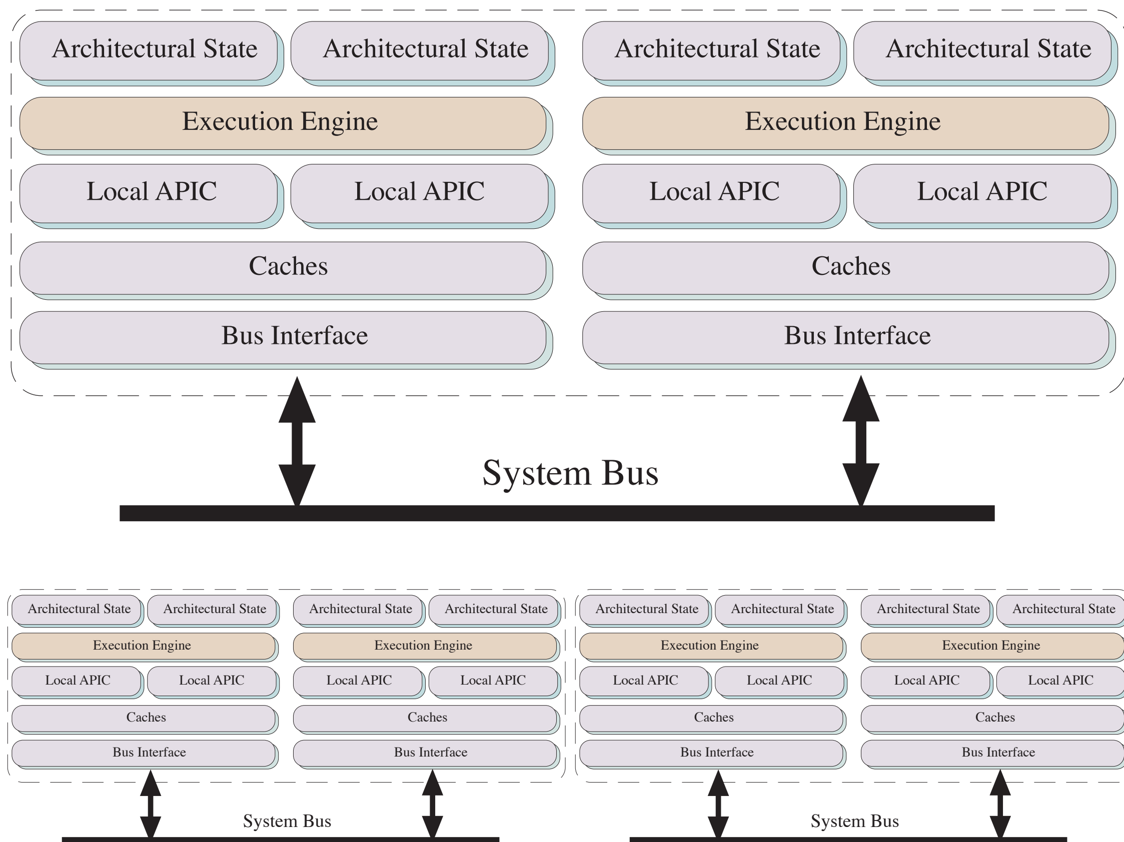


Figure 4-5: Intel Xeon 5100 Dual-Core Microprocessor Block Diagrams

© 2008 Rick Miller — All Rights Reserved

Three Aspects of Processor Architecture

There are 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 derives 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

Feature set implementation primarily determines how a processor's functionality is arranged and executed in hardware. How does the processor implement the feature set? Is it a Reduced Instruction Set Computer (RISC) or a 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. As processor power increases, the size of machine instructions grows as well, making it extremely difficult for programmers to deal directly with machine code.

From Machine Code To Assembly Language

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 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 C# add yet another layer of abstraction. C#, 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. 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.

© 2008 Rick Miller — All Rights Reserved

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 stored information 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, USB flash drives, 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 shown in Figure 4-6.

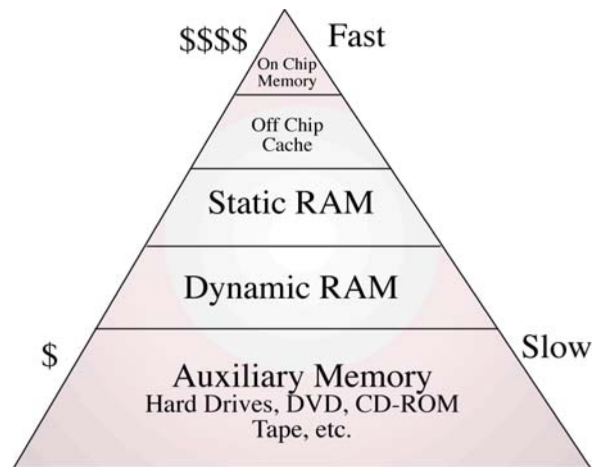


Figure 4-6: Memory Hierarchy

The job of a computer system designer with regards to memory subsystems is to make them perform as if all the memory they contained were fast and expensive. Utilizing cache memory to store frequently used data and instructions and buffering disk reads into memory 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 searched first by the processor for any requested data or instruction. If it's not there, a performance penalty occurs in the form of longer overall access times required to retrieve the information from a slower memory source. As chip densities grow, more cache memory will be located on the processor, thus improving overall processing times.

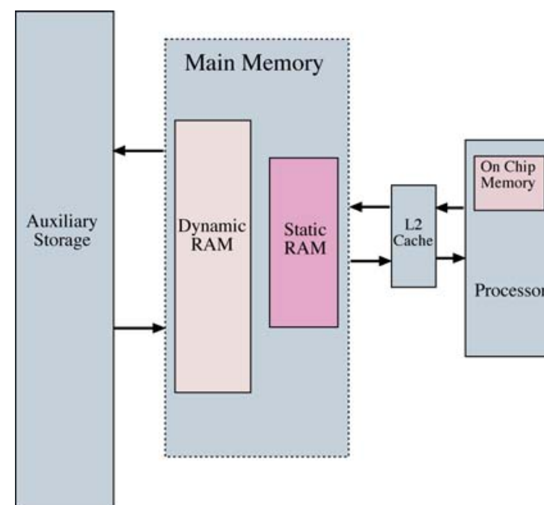


Figure 4-7: Simplified Memory Subsystem Diagram

Bits, Bytes, Words

Program code and data are stored in main memory as electronic voltages. Since I'm talking about digital computers, the voltage 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 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.)

© 2008 Rick Miller — All Rights Reserved

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 Xeon processor. 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 64 bits wide and its processor's registers are 64-bits wide, then the word size would be 8 bytes long (*64 bits / 8 bits = 8 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 64 bits wide.

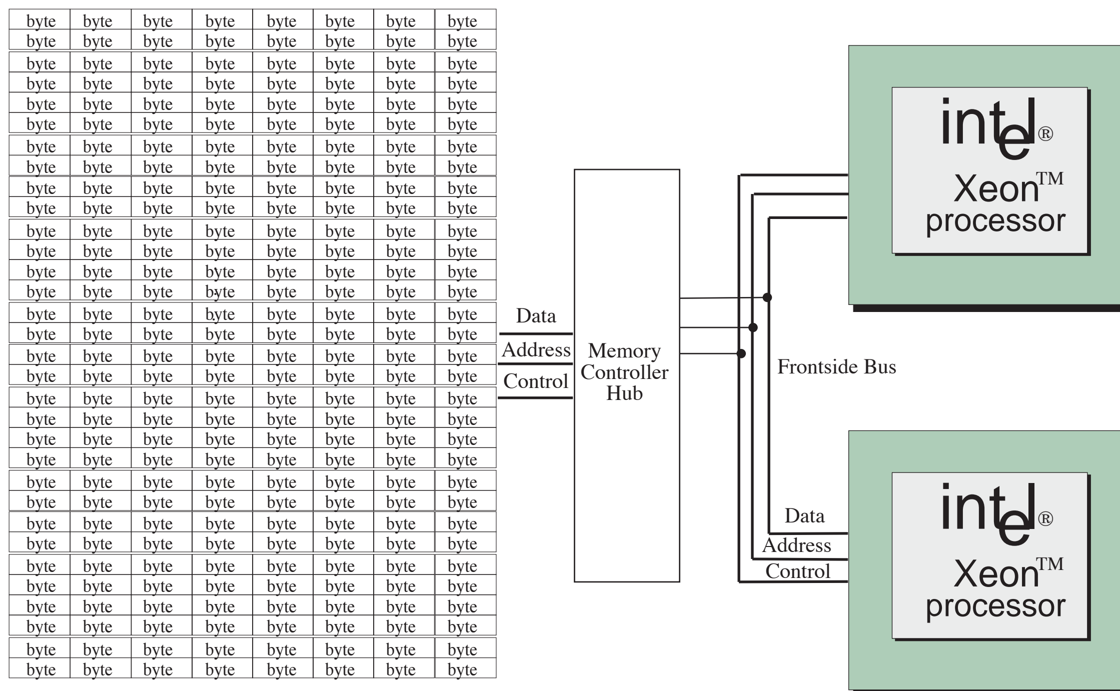


Figure 4-8: Simplified Main Memory Diagram

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 either a natural boundary or other type of boundary. For example, on a Xeon system, the contents of memory assigned to instances of structures are 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, and so on. Individual data elements not belonging to structures are usually aligned to eight-byte boundaries.

© 2008 Rick Miller — All Rights Reserved

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 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 if you are a human, but 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 C# 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 that the C# code a programmer writes must be translated from source code into machine instructions recognizable by a particular processor. This translation is effected by running a compiler that converts the C# code into an intermediate language that is then executed by the C# Common Language Runtime (CLR) Environment.

To a C# 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 methods to manipulate these object 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 first be loaded into main memory. The processor must then fetch the address of its first instruction, at which point execution begins. In the early days of computing, programs were coded into computers by hand and then executed. Nowadays, all of 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 shortly, 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 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 types of processors implement the processing cycle differently, but generally all processors carry out these four processing steps in some form or another. The processing cycle is depicted in Figure 4-9.

© 2008 Rick Miller — All Rights Reserved

Fetch

In the Fetch step, the processor reads an instruction from memory and presents it to its decode section. If cache memory is present, it is checked first. If the requested memory address contents resides in the cache, the read operation executes quickly. Otherwise, the processor must wait while the data is loaded from the slower main memory.

Decode

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

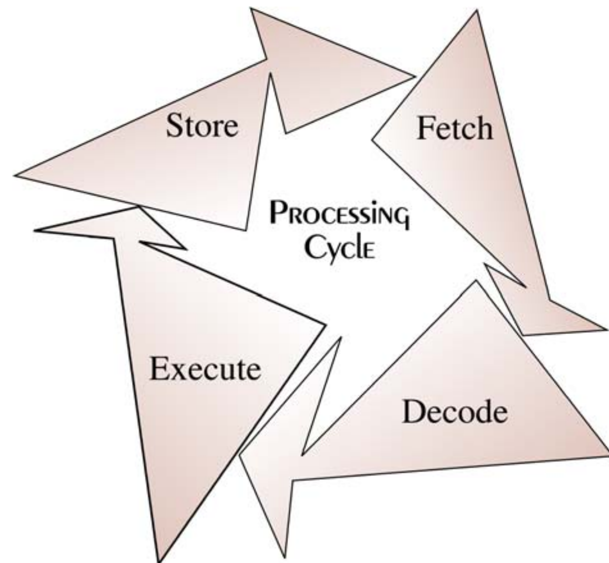


Figure 4-9: Processing Cycle

Execute

If the fetched instruction is successfully decoded as 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

When an instruction executes, the results, 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 these 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. When it fails to decode into a proper instruction, the computer halts.

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:

 © 2008 Rick Miller — All Rights Reserved

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.cs

```
1    using System;
2
3    public class DumbSort{
4      public static void Main(String[] args){
5        int[] a = {1,10,7,3,9,2,4,6,5,8,0,11};
6
7        int innerloop = 0;
8        int outerloop = 0;
9        int swaps = 0;
10
11       for(int i=0; i<12; i++){
12         outerloop++;
13         for(int j=1; j<12; j++){
14           innerloop++;
15           if(a[j-1] > a[j]){
16             int temp = a[j-1];
17             a[j-1] = a[j];
18             a[j] = temp;
19             swaps++;
20           }
21         }
22       }
23
24       for(int i=0; i<12; i++){
25         Console.Write(a[i] + " ");
26       }
27
28       Console.WriteLine();
29       Console.WriteLine("Outer loop executed " + outerloop + " times.");
30       Console.WriteLine("Inner loop executed " + innerloop + " times.");
31       Console.WriteLine(swaps + " swaps completed.");
32
33     }
34   }
```

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 7, 8, and 9, respectively. Figure 4-10 gives the results from running the dumb sort test program.

© 2008 Rick Miller — All Rights Reserved

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 ordered list of integers:

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

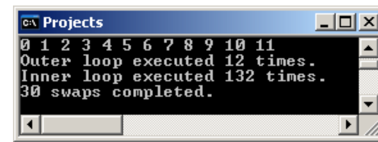


Figure 4-10: Dumb Sort Results 1

It appears that both the outer loop and inner loop execute 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 this time, because fewer swaps were necessary.

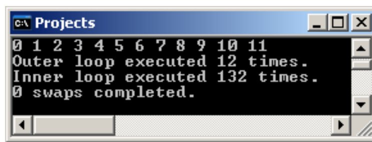


Figure 4-11: Dumb Sort Results 2

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 x 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. That is, 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.

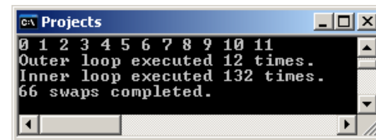


Figure 4-12: Dumb Sort Results 3

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.

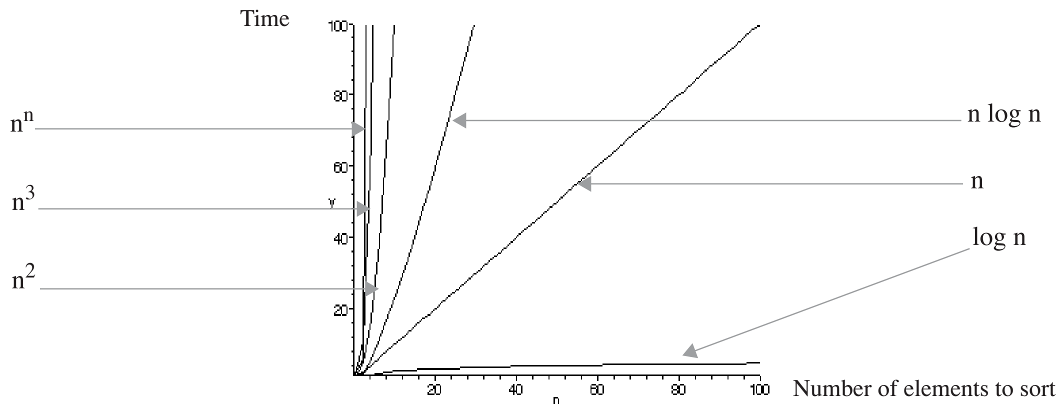


Figure 4-13: Algorithmic Growth Rates

© 2008 Rick Miller — All Rights Reserved

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 solutions.

Don't Reinvent The Wheel!

If you are new to programming, the best advice I can offer you is 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.

Virtual Machines And The Common Language Infrastructure

Figure 4-14 offers an overview of the C# compile and execute process.

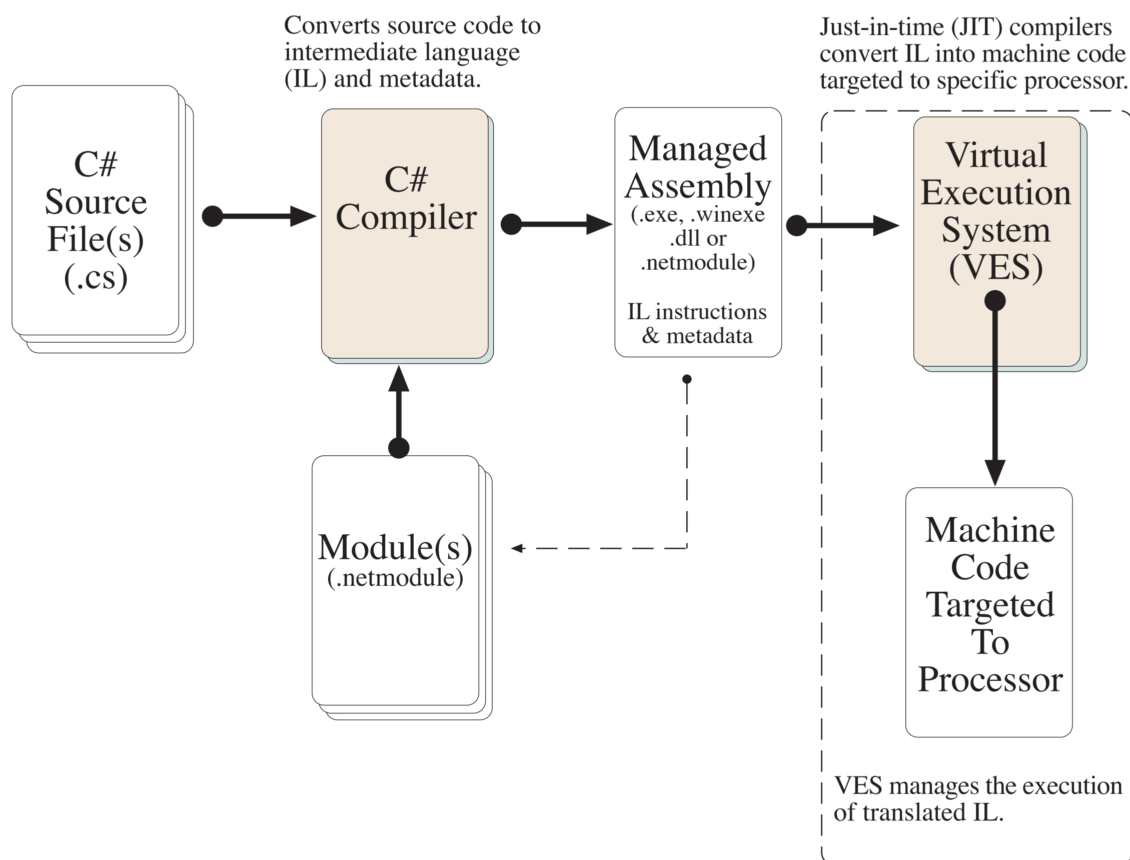


Figure 4-14: The C# Compile and Execution Process Overview

Referring to Figure 4-14 — the csc compiler compiles one or more C# source files into either a code module (.netmodule), a library (.dll), or one of two types of executable files: a console application (.exe) or a windows application (.winexe). Code modules are static IL code libraries whose code is referenced in your source file(s) and linked (added) to your project at compile time. A library, or dynamic link library (.dll), is a code module whose code is referenced in your source file(s) and loaded into the VES at application runtime. Hence the term "dynamic".

Executable files produced by the compiler can be loaded and executed by the VES. The VES executes and translates the IL instructions contained in the executable managed assembly. With the help of a JIT compiler, it produces machine code that is ultimately executed by the target processor. The JIT compiler is so named because it translates

 © 2008 Rick Miller — All Rights Reserved

IL into machine code as the IL instructions are executed by the VES. Blocks of compiled machine code are cached and tagged within the VES to prevent recompilation and to speed execution. Figure 4-15 shows what IL instructions look like.

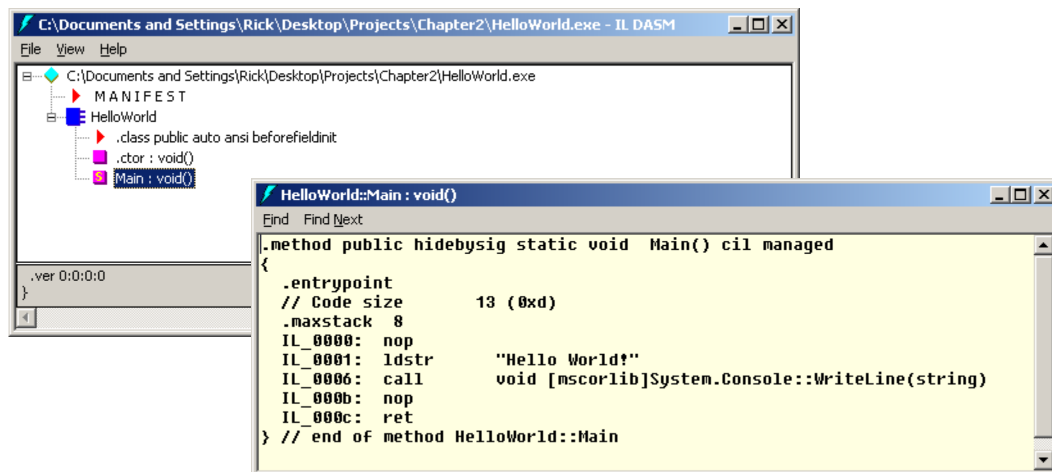


Figure 4-15: MSIL Disassembler Session Showing Main() Method IL Instructions

Referring to Figure 4-15 — the Microsoft Intermediate Language (MSIL) Disassembler tool provided by the Microsoft Windows Software Development Kit (SDK) is used to disassemble the HelloWorld.exe program used in Chapter 2. The IL instructions shown in the foreground window are those of the Main() method. These instructions are executed by the VES and translated into machine code when the HelloWorld.exe program executes.

Virtual Machines

The VES described in the previous section is a program (one or more software components acting in concert together) that executes IL instructions. In reality, the VES does more that simply execute IL instructions. I present a more detailed description of its responsibilities in the next section. Programs like the VES are referred to as *virtual machines*. The benefit of having C# target a virtual machine instead of a specific processor and operating system is the increased flexibility in the range of hardware and operating system environments on which C# programs can run.

A program written in any language whose compiler targets a specific processor must be recompiled for each different target processor on which the program must run. Not so with C#. Because the C# compiler targets a virtual machine, it can run on any computer platform that has on it an implementation of the VES. This cross-platform capability is made possible by an international standard known as ECMA - 335 Common Language Infrastructure (CLI) Partitions I to VI. So just what is this CLI and why should you care?

The Common Language Infrastructure (CLI)

ECMA - 335 specifies a CLI. As its name implies, the CLI specifies or lays down a set of rules that language makers, compiler makers, and virtual machine makers must follow if they want their languages and tools to run on different implementations of the CLI.

Four Parts Of The Common Language Infrastructure

The CLI provides architectural specifications for four areas: the Common Type System (CTS), metadata, the Common Language Specification (CLS), and the VES). Figure 14-16 graphically illustrates the relationship between these pieces of the CLI.

 © 2008 Rick Miller — All Rights Reserved

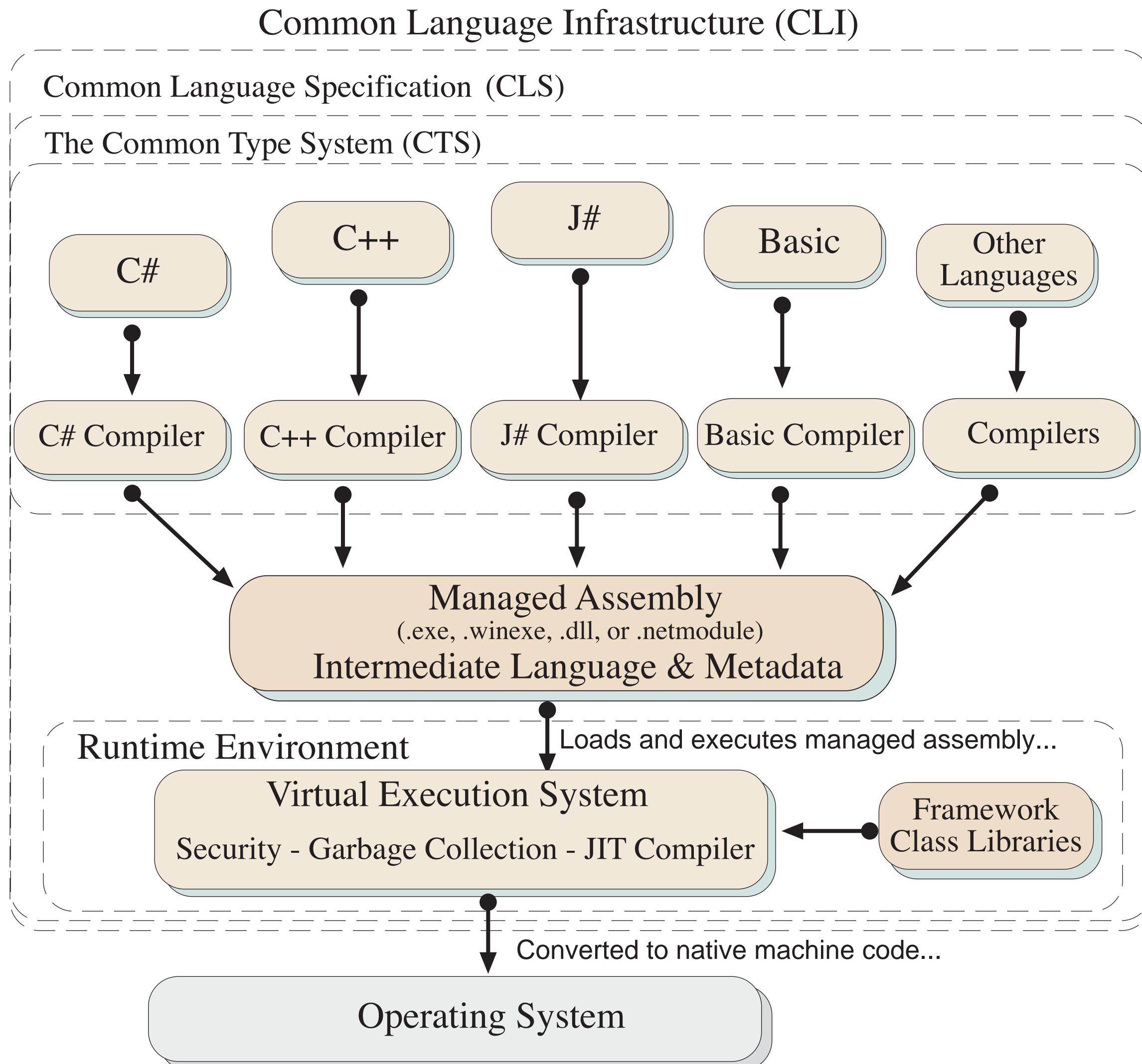


Figure 4-16: The Common Language Infrastructure Architecture

The Common Type System (CTS)

The CTS is the heart of the CLI. The CTS specifies a large set of types and operations common to many programming languages.

Metadata

The CLI uses metadata to describe and reference types defined by the CTS. Metadata can be thought of as data about data. Metadata is used by CLI tools and the Virtual Execution System (VES) to manipulate and manage IL code modules. Metadata is added to managed assemblies during the compilation process.

The Common Language Specification (CLS)

The CLS provides a set of rules that language and compiler implementors must follow to make their language interoperable with other CLI languages. Since languages share a CTS, modules generated by one language can be

 © 2008 Rick Miller — All Rights Reserved

used or referenced by programs written in another language. For example, Visual Basic.NET modules can be linked into and used by a program written in C#. This language interoperability is made possible by the CLS.

The Virtual Execution System (VES)

The VES executes *managed code* modules with the help of embedded metadata. **Note:** You can also write programs in C# that include what are referred to as *unmanaged code* segments. Unmanaged code segments allow direct access to the underlying operating system and hardware and thus tie a program to a specific platform.

The Cross Platform Promise

As long as you avoid unmanaged code, you can achieve some degree of cross-platform independence as Figure 4-17 illustrates, although in reality, Microsoft's implementation of the CLI (Microsoft .NET and the growing family of .NET compatible languages) will always, in my opinion, be well ahead of the competition.

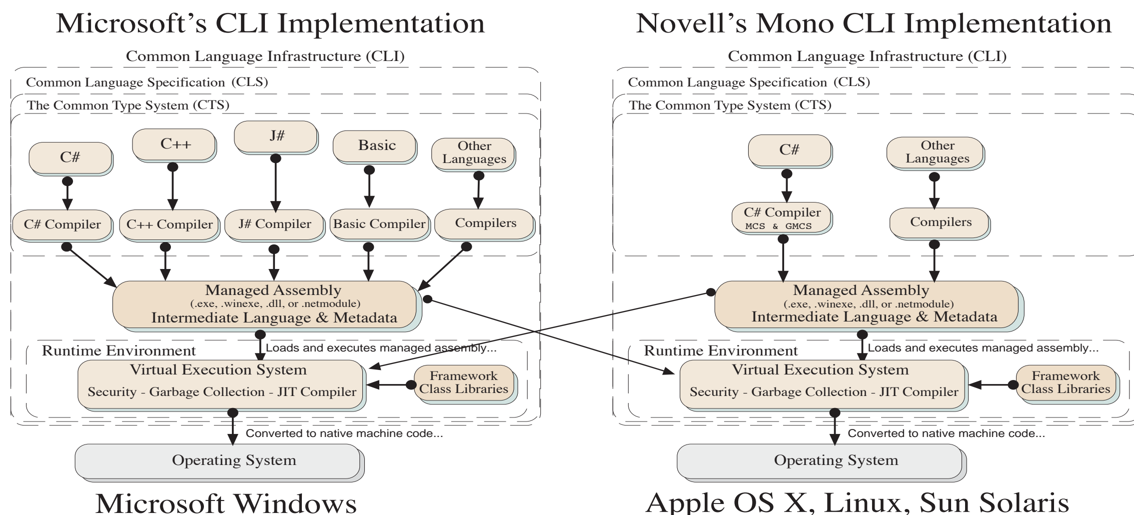


Figure 4-17: Managed Assemblies can be Executed on any System that Implements the Common Language Infrastructure

Referring to Figure 4-17 — as I write these words, the best attempt at a third party implementation of the CLI is the Novell's open-source Mono Project [http://mono-project.org]. As is shown in the diagram, the Mono project provides CLI implementations for Apple's OS X, various Linux platforms, Sun Solaris, and a few others not shown, including Microsoft Windows. Figure 4-18 shows the Robot Rat project of Chapter 3 executing in the Mono environment on a Macintosh G4 running Apple's OS X.

```
Terminal — mono
[Rick-Millers-Computer:Projects/Chapter3/final_version] swodog% mono RobotRat.exe
RobotRat Lives!


   RobotRat Control Menu

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

Figure 4-18: Chapter 3's Robot Rat Program Running in the Mono Environment on Apple OS X

© 2008 Rick Miller — All Rights Reserved

Figure 4-19 gives a simple diagram of Microsoft's .NET architecture. Compare this diagram with that of Figure 4-16. Applications created with .NET languages consisting entirely of managed code segments are referred to as managed applications. Applications that combine managed and unmanaged code segments are referred to as hybrid applications. Microsoft's implementation of the VES is called the Common Language Runtime (CLR). The CLR and the .NET class libraries are included with the .NET Framework.

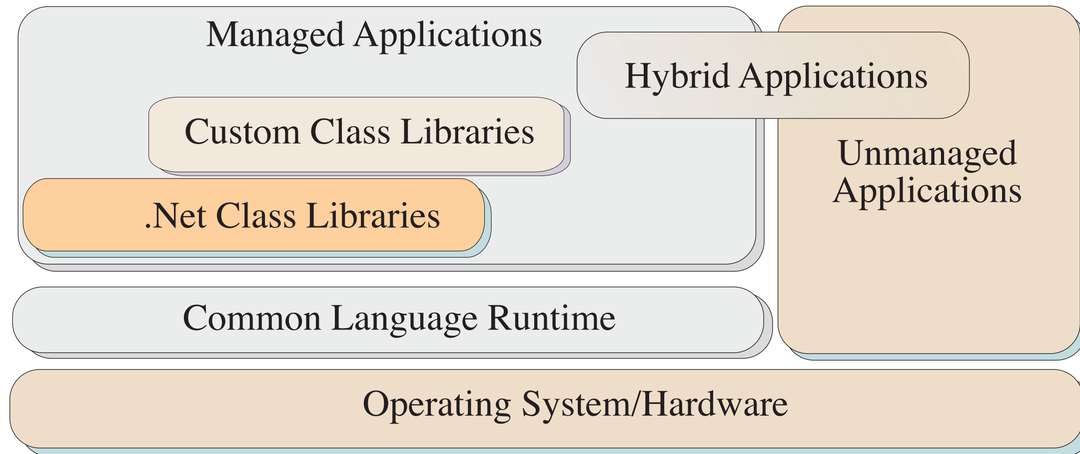


Figure 4-19: Microsoft .NET Architecture

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 C# help humans model extremely complex problems algorithmically. C# 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 but instead 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 to improve your code-writing skills.

Microsoft.NET is Microsoft's implementation of the Common Language Infrastructure (CLI) specification. Managed assemblies produced by the C# compiler contain descriptive metadata and execute within a Virtual Execution System (VES). The benefit of targeting a virtual machine is cross-platform execution.

C# is compiled into intermediate language (IL) instructions. The VES translates IL instructions into target processor machine code with the help of just-in-time (JIT) compilers.

If an assembly contains unmanaged code segments, then its cross-platform capabilities are limited.

Skill-Building Exercises

1. **Research Sorting Algorithms:** The second sorting algorithm listed on page 84 gives the steps for a Merge Sort. Obtain a book on algorithms, look for C# 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?

 © 2008 Rick Miller — All Rights Reserved

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 CLI:** Visit the ECMA website, download a copy of the CLI specification, and study the relationships between the CTS, metadata, the CLS, and the VES. [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

Suggested Projects

1. **Research Computer Systems:** Research your computer system. List all of its components including the type of processor. Go to the processor manufacturer's website 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 determine how the architecture of each implements the feature set.

3. **Disassemble a Managed Assembly:** The Microsoft Windows SDK is separate from the .NET Framework Runtime that you may have downloaded in Chapter 2. Download and install the SDK, and use the MSIL Disassembler to disassemble one of your C# project's executable file and inspect its intermediate language instructions.

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. What are the four steps of the processing cycle?

5. What, in your own words, does the term *algorithm* mean?

6. How does a processor's architecture serve to implement its feature set?

7. How can programmers access a processor's feature set?

8. What are the advantages of targeting a virtual machine vs. a physical processor? Can you think of any disadvantages?

9. What, if any, are the disadvantages of having unmanaged code segments in a C# program?

10. What is meant by the term just-in-time compiler?

 © 2008 Rick Miller — All Rights Reserved

References

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Intel Corporation Design Documentation, *IA-32 Intel Architecture Optimization Reference Manual*, Order Number: 248966-013US, April 2006

Intel Corporation Design Documentation, *Intel Xeon Processor with 512kb L2 Cache at 1.8 GHZ to 3 GHZ Datasheet*

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Apple Computer, Incorporated website [http://www.apple.com]

Notes

© 2008 Rick Miller — All Rights Reserved

Chapter 5

Contax T3 / Kodak Tri-X

Branches

Navigating .NET Framework Documentation

Learning Objectives

- *Use Microsoft Developer Network (MSDN) to search for .NET Framework Documentation*
- *State the definition of the term "Application Programming Interface" (API)*
- *List and describe the Base Class Libraries of the .NET Framework API*
- *Demonstrate your ability to navigate a class inheritance hierarchy*

© 2008 Rick Miller — All Rights Reserved

Introduction

When programming in C# or any .NET programming language, a lot of your work is already done for you in the form of the .NET Framework class library. The .NET Framework class library supplies interfaces, classes, and value types that serve as the foundation upon which all .NET applications are built, and provides advanced functionality to help you create feature-rich applications.

The .NET class library, also referred to as the .NET Framework application programming interface (API), is both a blessing and a curse. It's a blessing because just about every conceivable thing you would want to do in your programs, from creating and managing collections of objects to writing complex client-server networked database applications, is available for immediate use in the form of pre-existing interfaces and classes. It's a curse in that you must expend considerable effort in learning how to use these interfaces and classes in your programs.

The purpose of this chapter is to help you jumpstart your usage of .NET API classes by showing you how to look up API information on Microsoft's MSDN website and navigate class inheritance hierarchies. The successful and quick navigation of the .NET API reference material is a fundamental programming skill employed everyday by programmers around the world.

In fact, you will spend much more time learning how to use the .NET API than you will learning C# language fundamentals. This is due largely to the shear number of classes the API contains and partly because the API continuously evolves. But don't panic. You can get started programming complex, professional-looking C# applications with the help of only a small handful of API classes.

MSDN: The Definitive Source For API Information

The definitive source for .NET API information is the Microsoft Developer Network (MSDN) website [www.msdn.com]. If you don't have an internet connection but did buy Microsoft Visual Studio.NET or get the C#.NET Express Edition CD, then you can access the .NET API reference documentation that came with those products.

On the MSDN website, navigate to the .NET API reference section by following the links .NET Framework -> Class Library Reference. This will bring you to a page that looks similar to that shown in Figure 5-1.

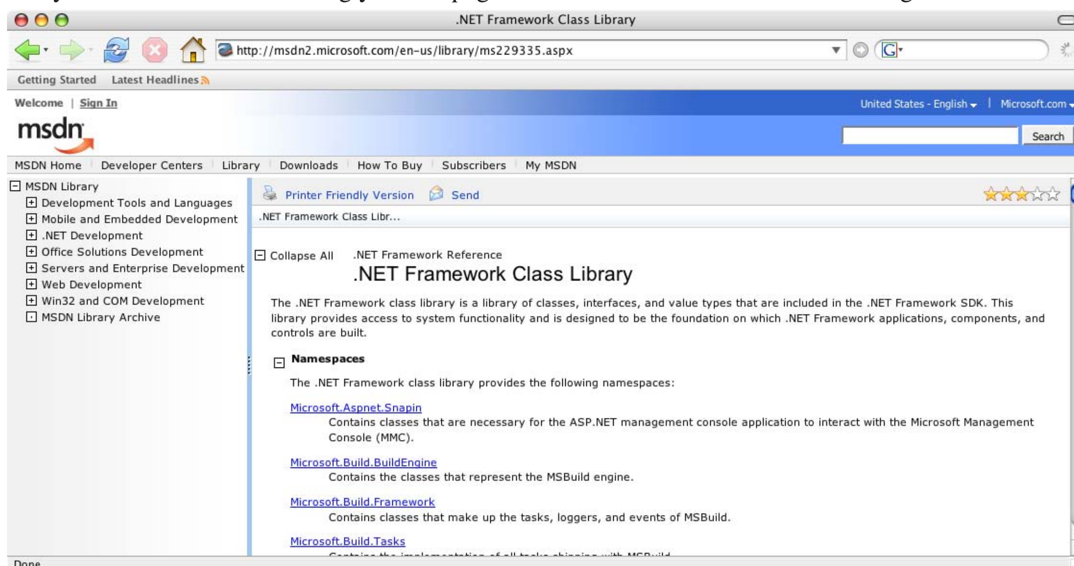


Figure 5-1: .NET Framework Class Library Reference Page

© 2008 Rick Miller — All Rights Reserved

All of the .NET Framework development information you can access from this page can be overwhelming, but for the purposes of this book, you only need concern yourself with a very small part of the API reference. In the left frame, click the .NET Development link to expand it as shown in Figure 5-2.

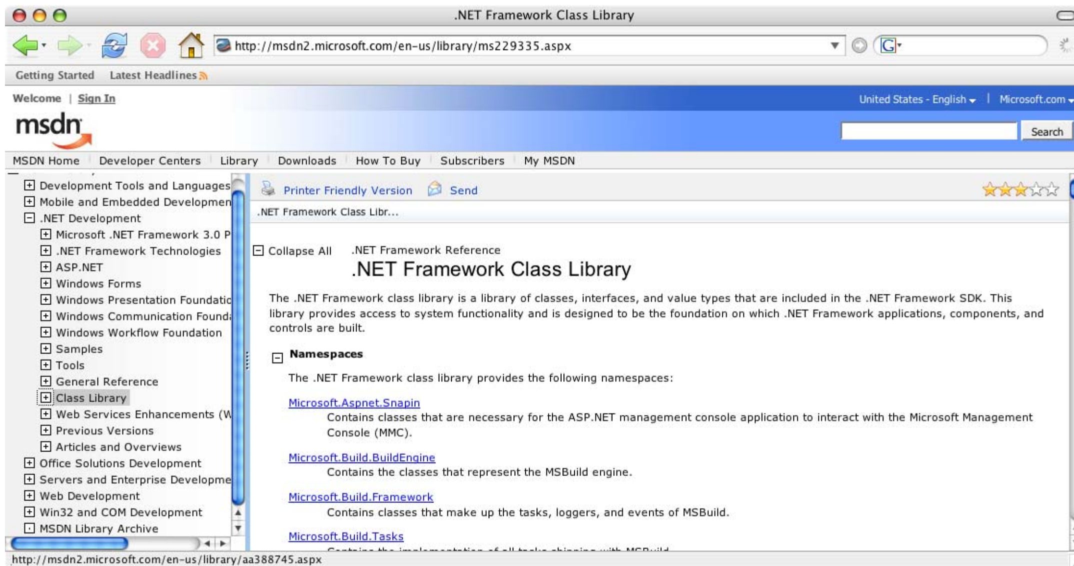


Figure 5-2: .NET Development Link Expanded and Class Library Link Highlighted

Again in the left frame, click the Class Library link to expand it, and scroll down until you find the System namespace as shown in Figure 5-3. It is in the Class Library section of the .NET API reference that you will spend most of your time researching and referencing the value types, interfaces, and classes found in the System namespace and its sub namespaces.

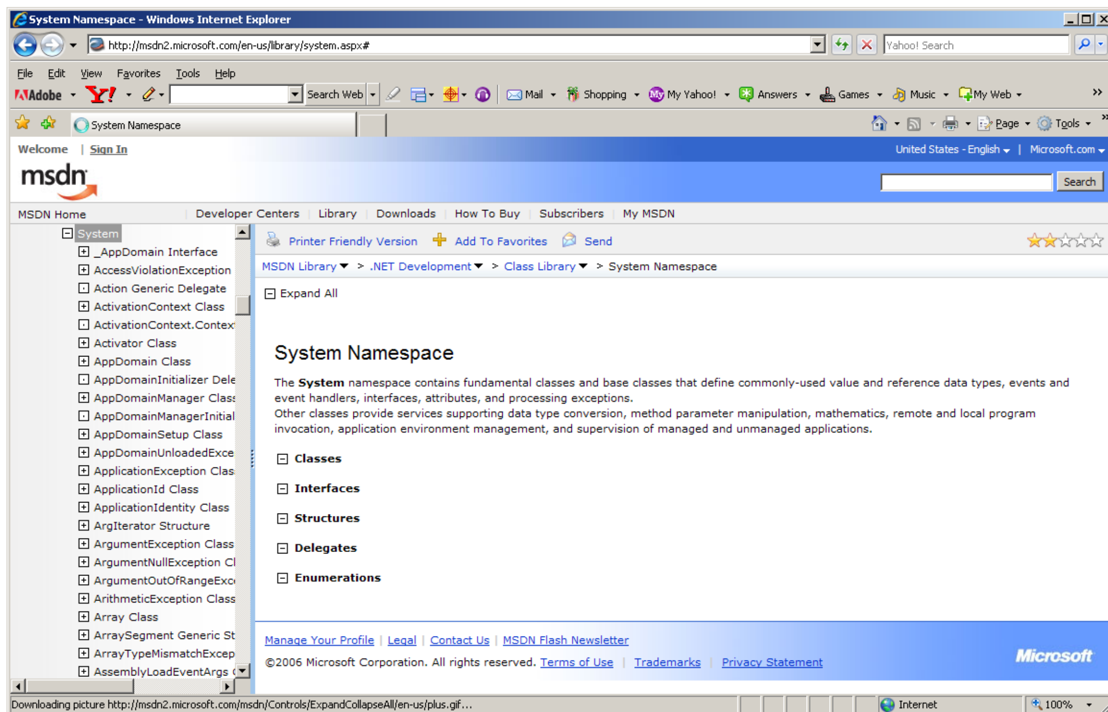


Figure 5-3: Class Library Link Expanded and System Namespace Highlighted

 © 2008 Rick Miller — All Rights Reserved

Discovering Information About Classes

If you look closely at Figure 5-3 you'll see in the right frame under System Namespace a short paragraph describing its contents. Below that you'll see several subheadings (collapsed in the figure). The subheadings include Classes, Interfaces, Structures, Delegates, and Enumerations. The System namespace contains a lot of stuff; indeed, the System namespace is the primary library in the .NET Framework. To get a feel for how to navigate API information, let's take a look at the String class. Click the Classes subheading to reveal all the System namespace classes. Scroll down until you find the String class. Click the String class link to open a window that looks similar to that shown in Figure 5-4.

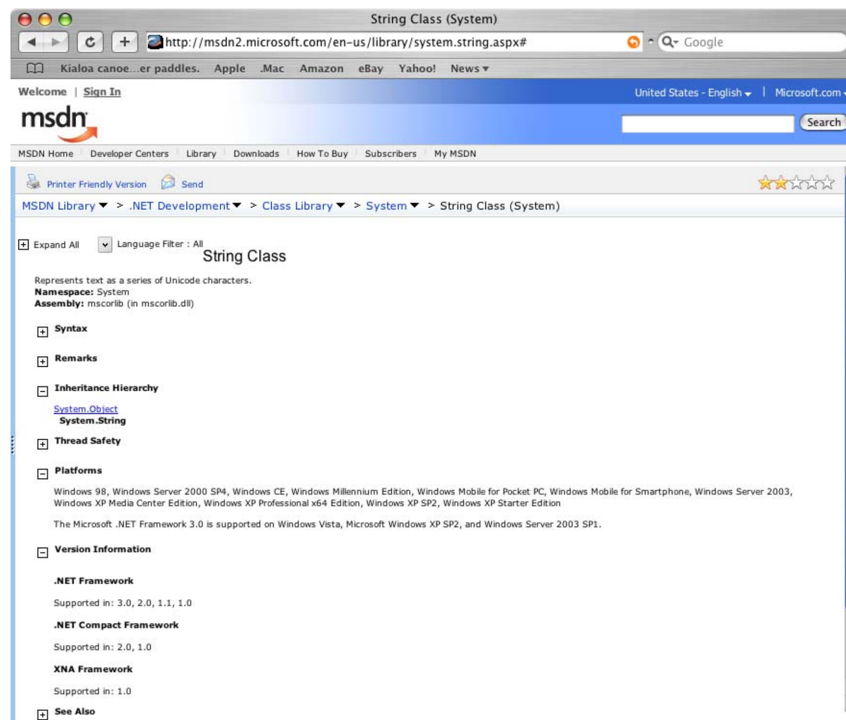


Figure 5-4: String Class API Reference Overview Page

General Overview Page

Referring to Figure 5-4 — What you are looking at here is the class information overview page. The overview page contains a lot of good general information arranged in subheadings or sections. I have collapsed a few of the subheadings for the purpose of this figure, but note that the API reference pages normally load with all subheadings fully expanded.

The Syntax section shows the class declaration. This is handy when you are trying to navigate a class's inheritance hierarchy, which I will discuss in greater detail later in this chapter.

The Remarks section contains information on how to use the class, and discusses issues you should be aware of when using the class in your code.

The Inheritance Hierarchy section shows you what classes this particular class extends or inherits. As you can see, the String class extends Object. **Note:** The Inheritance Hierarchy section does not show what interfaces the String class implements. For that you need to examine the Syntax section.

The Thread Safety section offers some advice on using the class in multithreaded programs.

The Platforms and Version Information sections show the Microsoft Windows operating system platforms that support this class and the different .NET Framework versions in which it appears. Not all .NET Framework classes, interfaces, etc., are supported on all platforms.

 © 2008 Rick Miller — All Rights Reserved

The See Also section provides links to other elements within the .NET Framework that share a relationship with this entry. For example, here you would find links to the numerous interfaces implemented by the String class.

Class Member Page

A class contains members. These members can include constructor methods (constructors), fields, properties, and methods. What you'll see on the members page is information about public members. Public members are those class members that you have access to when you use objects of this type in your code. If a class can be extended, meaning it's not sealed, you'll see its protected members as well. Figure 5-5 shows the String Members page with the subheadings collapsed.

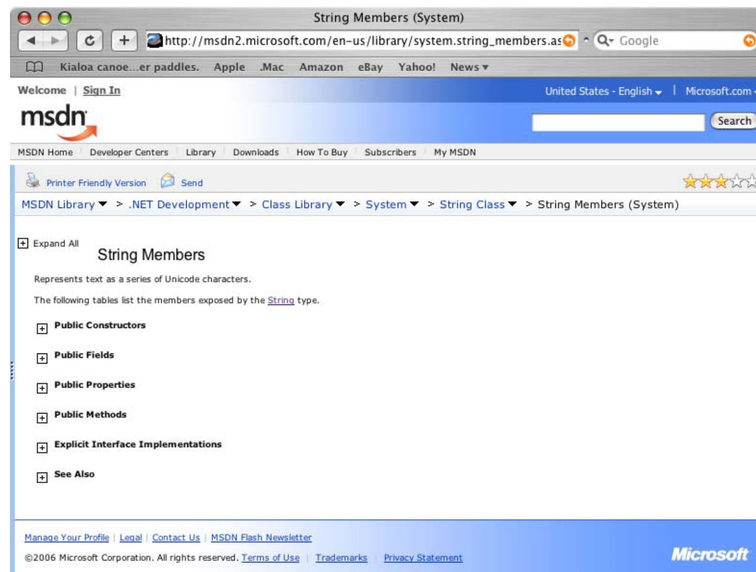


Figure 5-5: String Members Page

The Public Constructors section shows you the list of public constructor methods the class contains and notes on how to use them. Figure 5-6 shows a partial listing of the String class's Public Constructors section.

The Fields section lists any public fields the class makes available for use. The Properties section lists any public properties the class may have. The same hold for the Methods section. A partial listing of the String class's Methods page is shown in Figure 5-7.

Click a particular method's link to get more information. For example, scroll down the String Methods page, find the SubString method, and click its link. This will take you to the String.SubString Method page, as is shown in Figure 5-8.

Referring to Figure 5-8 — notice there are two versions of the SubString method. This means the SubString method has been overloaded to perform a similar operation in two different ways. Click one of the method links to learn more about how to use that particular method version, as Figure 5-9 illustrates.

Referring to Figure 5-9 — notice there are several subheadings on the method details page. The Syntax section shows how the method is declared in several different .NET programming languages. The Exceptions section lists and describes any exceptions the method may throw if something goes wrong when it's called. The Remarks section provides a few words of guidance on the method's use. The Platforms and Version sections contain the same types of information as they do on other API pages.

The Example section is the one part of the page you will be most interested in studying. It provides examples of how to use the method in several different programming languages. Figure 5-10 shows the Examples section expanded to reveal the C# code section.

 © 2008 Rick Miller — All Rights Reserved

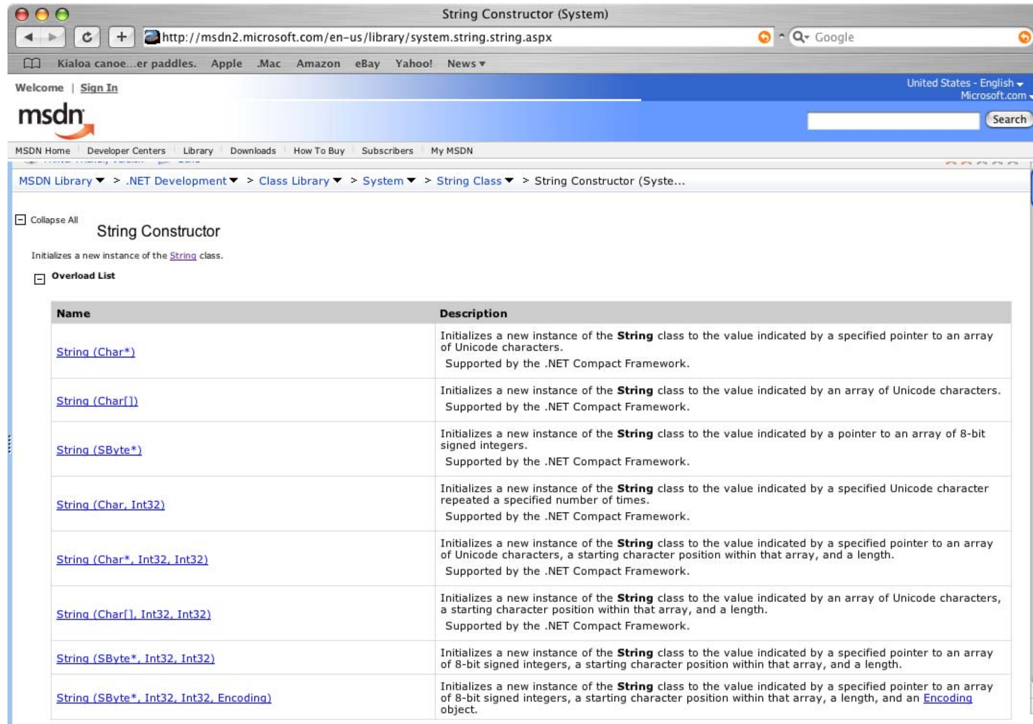


Figure 5-6: String Class's Public Constructors Partial Listing

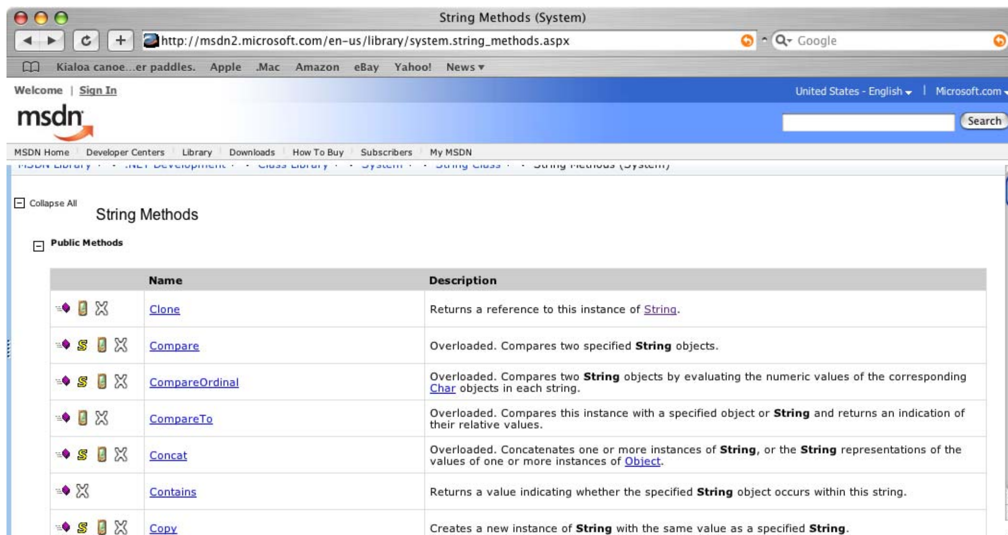


Figure 5-7: String Class's Methods Page Partial Listing

Getting Information On Other Class Members

Additional information on class fields, properties, constructors, etc., can be obtained in the same way as information on class methods; Just follow the links. At this time, you may find it extremely helpful just to wander around the .NET Framework documentation. Explore the System namespace and its many sub namespaces and see what types of classes and interfaces they contain. Don't be put off by not understanding what it is you are looking at. The important thing to do is to simply get familiar with .NET Framework documentation. Doing so will pay huge dividends in the very near future.

 © 2008 Rick Miller — All Rights Reserved

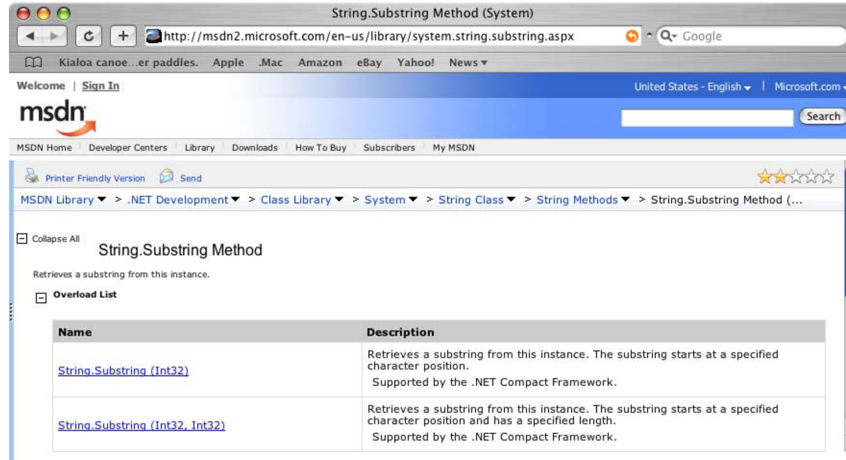


Figure 5-8: String.SubString Method Page

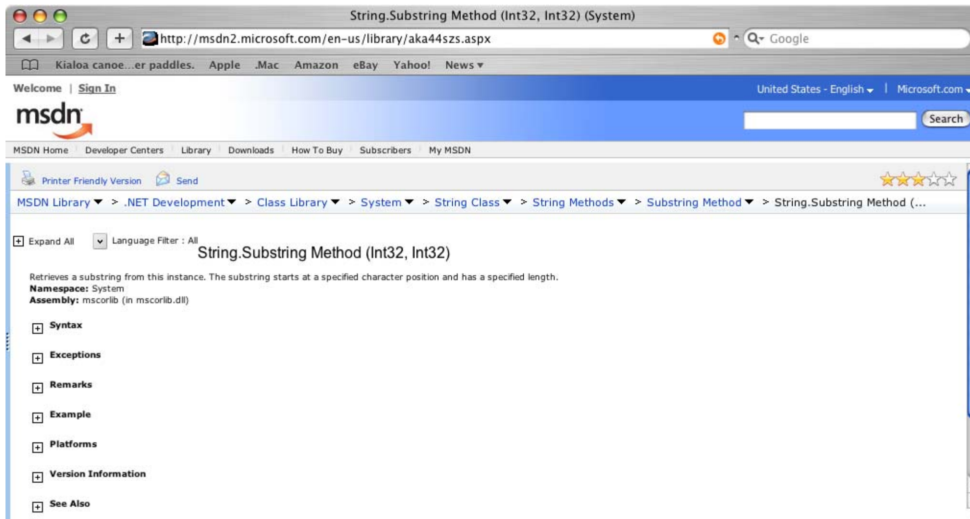


Figure 5-9: String.SubString Page with Collapsed Subheadings

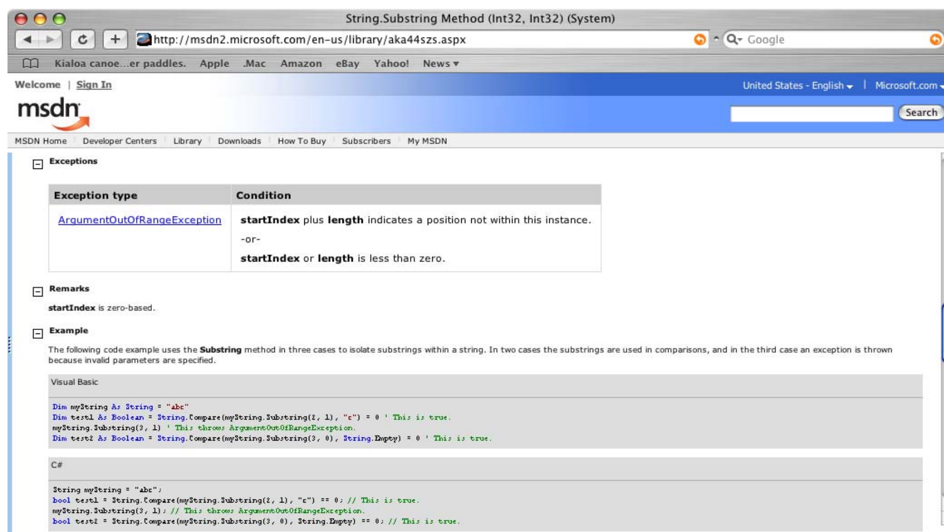


Figure 5-10: String.SubString Example Section Expanded Showing Example Code

© 2008 Rick Miller — All Rights Reserved

Quick Review

The MSDN is the definitive source for .NET Framework API information. You'll spend most of your time in the Class Library section researching the many classes, interfaces, and value types that appear in the System namespace and its many subnamespaces.

The Base Class Libraries (BCL)

Regardless of what type of C# application you build, your program will fundamentally depend upon a small core of classes that belong to the Base Class Libraries (BCL). The Base Class Libraries include the contents of the namespaces listed Table 5-1.

Namespace	Description
System	This is a fundamental namespace that includes all value type structures, the String class, math functionality, the DateTime class, and much, much more.
System.CodeDom	Supports the ability to dynamically create and execute code.
System.Collections	Contains interfaces and classes that let you manipulate collections of objects. Includes data structures such as lists, hashtables, and dictionaries.
System.Diagnostics	Provides the ability to diagnose the performance of your application.
System.Globalization	Provides the capability to internationalize your application.
System.IO	Supports file, console, serial port, and interprocess input and output.
System.Resources	Provides classes and interfaces that allow you to store and manipulate culture-specific application resources.
System.Text	Provides the capability to manipulates sequences of ASCII, Unicode, UTF-7, and UTF-8 character encodings.
System.Text.RegularExpressions	Supports the use of regular expressions in your .NET applications.

Table 5-1: Base Class Library (BCL) Namespaces

In addition to these libraries, this book will draw heavily from the namespaces shown in Table 5-2.

Namespace	Description
System.Collections.Generic	Provides many different types of generic collection classes.
System.Data System.Data.SQL	Provides the capability to write applications that access a relational database using ADO.NET and Structured Query Language (SQL)
System.Drawing	Provides graphics drawing and manipulation classes.
System.Net System.Net.Sockets	Provides classes and interfaces used to write networked applications.
System.Runtime.Remoting	Provides classes and interfaces necessary to write distributed applications that call methods on remote objects.
System.Runtime.Serialization	Supports the serialization and deserialization of objects.

Table 5-2: Additional .NET Libraries Used Heavily In This Book

© 2008 Rick Miller — All Rights Reserved

Namespace	Description
System.Threading	Provides multithreaded application support.
System.Windows.Forms System.Windows.Forms.Layout	Provides a large collection of classes and interfaces used to create Windows Graphical User Interface (GUI) applications.

Table 5-2: Additional .NET Libraries Used Heavily In This Book

Quick Review

The Base Class Libraries include those namespaces, and the classes they contain, that serve as the fundamental building blocks of the .NET Framework.

Navigating An Inheritance Hierarchy

In this section, I want to show you how to navigate an inheritance hierarchy. I'll use the String class as an example. Figure 5-11 shows a Unified Modeling Language (UML) diagram for the String class's inheritance hierarchy.

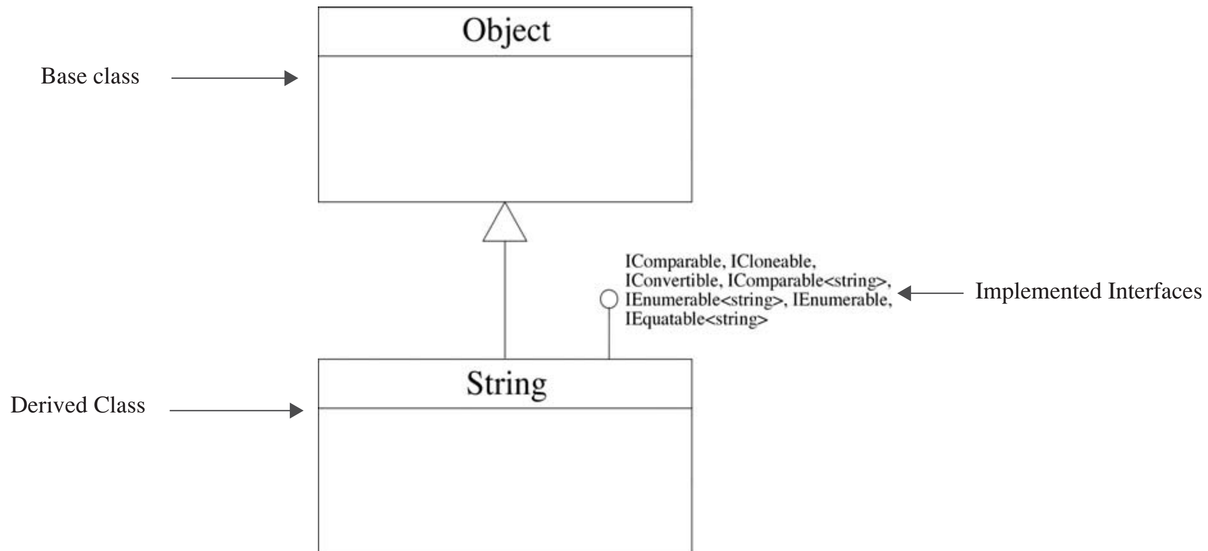


Figure 5-11: String Class Inheritance Hierarchy

Referring to Figure 5-11 — the UML diagram shown is referred to as a "class" diagram. A class diagram shows static relationships between classes, interfaces and other system artifacts. In this case, the String class inherits from the Object class. How do we know this? By referring to the String class overview page available in the .NET Framework documentation. The inheritance hierarchy is shown in the Inheritance Hierarchy section as was shown in Figure 5-4. Unfortunately, the Inheritance Hierarchy section only gives part of the picture. You need to study the Syntax section of the class overview page to learn what interfaces a class implements and any attributes, such as *SerializableAttribute* or simply *Serializable*, it supports. The String class declaration as given in the Syntax section of the String class documentation page appears in Example 5-1.

5.1 String Class Declaration

```
1    [SerializableAttribute]
2    [ComVisibleAttribute(true)]
3    public sealed class String : IComparable, ICloneable, IConvertible, IComparable<string>,
                         IEnumerable<string>, IEnumerable, IEquatable<string>
```


© 2008 Rick Miller — All Rights Reserved

Referring to Example 5.1 — the String class declaration does not explicitly inherit from Object. Rather, all C# types (*i.e.*, reference types [classes] and value types [structures]) implicitly extend Object, although value type structures implicitly extend System.ValueType, which extends System.Object, and behave differently from reference types.

OK. So what's the use of tracking down the base classes and interfaces of a class? Simple: class behavior is the sum of all behaviors inherited from base classes, from interface implementations, or applied attributes. (*e.g.*, SerializableAttribute is an example of an attribute.) To fully understand all that a particular class can do, you must navigate up the inheritance hierarchy, visit each base class, and in turn visit each interface to learn what it means to provide an implementation for it.

In short, a String is an Object. Any methods declared public in the Object class can also be called on a String object. A String object is not only serializable and comvisible, it is also comparable, cloneable, convertible, enumerable, and equatable.

Do not panic if you don't know yet what all this mumbo jumbo means. By Chapter 11 it will all start to make perfect sense. However, it would still be a good exercise to visit the String class now and follow the links to all its related interfaces. Also, visit the Object class and see what it has to offer.

Quick Review

Trace a class's inheritance hierarchy to discover its complete range of functionality.

Beware Obsolete APIs

As the .NET Framework evolves, there will be times when some of what came before will be rendered obsolete. Though using an obsolete API component does not immediately spell disaster, it's a good idea to avoid using them when possible for the sake of forward compatibility. You can get a listing of obsolete API members by visiting the MSDN home page and following these links: Common Language Runtime -> Reference -> .NET Framework V2.0 Obsolete API List. Figure 5-12 offers a partial listing of obsolete API components by Namespace.

Figure 5-12: Obsolete .NET Framework Version 2.0 API Partial Listing by Namespace

 © 2008 Rick Miller — All Rights Reserved

Summary

The Microsoft Developer Network (MSDN) is the definitive source for .NET Framework API information. You'll spend most of your time there in the Class Library section researching the many classes, interfaces, and value types that appear in the System namespace and its many subnamespaces.

The Base Class Libraries (BCL) include those namespaces, and the classes they contain, that serve as the fundamental building blocks of the .NET Framework.

Trace a class's inheritance hierarchy to discover its complete range of functionality.

Skill-Building Exercises

1. **API Drill:** Visit the MSDN homepage and explore the many links provided.

2. **API Drill:** Explore the System namespace. Write down a list of all classes, interfaces, exceptions, events, and delegates you find there. Research the functionality each provides and write a brief description of that functionality using your own words.

3. **API Drill:** Explore the System.Text namespace. Write down a list of all classes, interfaces, exceptions, events, and delegates you find there. Research the functionality each provides and write a brief description of that functionality using your own words.

4. **API Drill:** Explore the System.Collections namespace. Write down a list of all classes, interfaces, exceptions, events, and delegates you find there. Research the functionality each provides and write a brief description of that functionality using your own words.

5. **API Drill:** Explore the System.Collections.Generic namespace. Write down a list of all classes, interfaces, exceptions, events, and delegates you find there. Research the functionality each provides and write a brief description of that functionality using your own words.

6. **API Drill:** Explore the System.IO namespace. Write down a list of all classes, interfaces, exceptions, events, and delegates you find there. Research the functionality each provides and write a brief description of that functionality using your own words.

7. **API Drill:** Explore the System.Net namespace. Write down a list of all classes, interfaces, exceptions, events, and delegates you find there. Research the functionality each provides and write a brief description of that functionality using your own words.

8. **API Drill:** Explore the System.Threading namespace. Write down a list of all classes, interfaces, exceptions, events, and delegates you find there. Research the functionality each provides and write a brief description of that functionality using your own words.

9. **API Drill:** Explore the System.Drawing namespace. Write down a list of all classes, interfaces, exceptions, events, and delegates you find there. Research the functionality each provides and write a brief description of that functionality using your own words.

10. **API Drill:** Explore the System.Windows.Forms namespace. Write down a list of all classes, interfaces, exceptions, events, and delegates you find there. Research the functionality each provides and write a brief description of that functionality using your own words.

© 2008 Rick Miller — All Rights Reserved

Suggested Projects

1. **Navigate Inheritance Hierarchy:** Navigate the inheritance hierarchy for the System.String class. Follow the links for its base class and all implemented interfaces. Write down the functionality provided by each interface. Make a note of any overridden methods found in the class.

2. **Navigate Inheritance Hierarchy:** Navigate the inheritance hierarchy for the System.Int32 structure. Follow the links for its base class and all implemented interfaces. Write down the functionality provided by each interface. Make a note of any overridden methods found in the class.

3. **Navigate Inheritance Hierarchy:** Navigate the inheritance hierarchy for the System.DateTime structure. Follow the links for its base class and all implemented interfaces. Write down the functionality provided by each interface. Make a note of any overridden methods found in the class.

4. **Navigate Inheritance Hierarchy:** Navigate the inheritance hierarchy for the System.Windows.Forms.Button class. Follow the links for its base classes and all implemented interfaces. Write down the functionality provided by each interface. Make a note of any overridden methods found in the class.

5. **Navigate Inheritance Hierarchy:** Navigate the inheritance hierarchy for the System.Convert class. Follow the links for its base class and all implemented interfaces. Write down the functionality provided by each interface. Make a note of any overridden methods found in the class.

Self-Test Questions

1. Where can you find the most recent version of .NET Framework documentation?

2. What types of information can you find on a class overview page?

3. How would you find some example code showing the use of a particular class method?

4. What's the purpose of knowing how to navigate a class inheritance hierarchy?

References

Microsoft Developer Network website [http://msdn.com]

© 2008 Rick Miller — All Rights Reserved

Notes

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Part II: Language Fundamentals

©2008 Rick Miller — All Rights Reserved

©2008 Rick Miller — All Rights Reserved

Chapter 6

Voigtlander Bessa-L / 15mm Super Wide-Heliar

Chipotle – Rosslyn, VA

Simple C# Programs

Learning Objectives

- *State the required parts of a simple console application*
- *State the definition of the terms: "Application", "Assembly", and "Module"*
- *State the purpose of the Main() method*
- *Describe the differences between the four different versions of the Main() method*
- *State the purpose of the "using" directive*
- *Describe the differences between value types and reference types*
- *State the purpose of statements, expressions, and operators*
- *State the purpose of the "new" operator*
- *Apply the "new" operator to dynamically create objects in memory*
- *List and describe the use of the C# operators*
- *List and describe the use of the C# reserved keywords*
- *Demonstrate your ability to create simple C# programs*
- *Demonstrate your ability to compile C# programs using the command-line compiler*

© 2008 Rick Miller — All Rights Reserved

Introduction

This chapter lays a solid foundation for the understanding of the material contained within the remaining chapters of this book. Here you will learn the fundamental concepts crucial to building C# applications. As the old adage goes, you must learn to crawl before you can walk. Soon you will be running. But in the meantime, you will begin to wonder whether you will ever get off the floor!

Try as I may to make the material contained here easy to understand and free of confusing concepts, I am hindered in doing so by the very nature of the C# language. For example, the simplest program you can write in C# must be contained within a class. Thus, the concept of a class is forced upon you when it would be nice to delay its discussion until later.

The primary challenge facing both students and teachers of a modern object-oriented programming language like C# is the multitude of complexities presented by both the language itself and its accompanying collection of framework classes, referred to in the case of C# as the .NET Framework Application Programming Interface (API). I will mitigate this complexity in this chapter by keeping the example programs small and concise, and by limiting the use of .NET Framework API classes to those required for simple console input and output.

I will focus my efforts on helping you understand the C# type structure and understanding the differences between value types and reference types. I will explain to you the purpose of the Main() method, and show you how to use value type and reference type objects within a Main() method. I will also show you how to use variables and constants in simple programs. I will then discuss the C# language operators and demonstrate their use.

If you are completely new to programming, even the material I talk about in this chapter can be intimidating. Be patient and keep at it. A keen grasp of the fundamentals pays big dividends when you start to tackle more complex concepts.

What Is A C# Program?

When I say to you, “Write a program in C# to do this or that...,” what do I mean? There are many answers to this question, and all of them are correct. Each depends on the complexity of the problem being solved and the particular approach you might take towards its solution. For example, as you will soon see later in this chapter, if I ask you to write a program that adds two numbers and displays the result on the screen, you can write this program as a console application contained in one class. The effort spent analyzing the problem (*i.e.*, adding two numbers and displaying the sum) will be minimal.

Another approach to writing the simple adding program might involve the use of graphical user interface (GUI) components so that users could enter the numbers to be added in a familiar window interface. This version of the program could be written either as one class or as multiple classes. It depends on how you approach the design of the program. The GUI version of the program also would use more of the .NET API classes to create the window and handle user interactions within the interface.

The approach you take to the design of a program depends largely on how much you know about designing programs. As you progress through this book, you will learn the C# language and program design concepts hand-in-hand. At first you will see examples of simple, one-class programs. As you are introduced gradually to object-oriented programming concepts, your knowledge of program design will increase and you will be able to build more complex programs.

So, when I say, “Write a program in C# to do this or that..., ” what you do might be as simple as creating one class and adding a few lines of code to do a simple operation. This simple program will be contained in one source file. For more complex programming projects, you may need to spend considerable time analyzing the problem at hand and designing a suitable solution using object-oriented analysis, design, and programming techniques. The resulting program may be spread across multiple files. But first, you must learn to crawl.

© 2008 Rick Miller — All Rights Reserved

A Simple Console Application

In this section you will learn how to build simple, one-class console applications. This one-class program design will serve as the basis for demonstrating many fundamental concepts throughout this and several later chapters.

Definition Of Terms: Application, Assembly, Module, and Entry Point

An *application* is an *assembly* that has an *entry point*. In the words of the Common Language Infrastructure (CLI) standard: "An assembly is a configured set of loadable code modules and other resources that together implement a unit of functionality," and, "A module is a single file that can be executed by the virtual execution system (VES)." (See Chapter 4 for a discussion of the VES). These are nice definitions, but are rather technical and somewhat misleading — and in dire need of explanation.

The entry point is a fancy name for where a program can begin execution. When an application is loaded into the VES, it has to start execution at some point in the code. This point is referred to as the entry point. The entry point is nothing more than a special method known as the Main() method. I will talk more about the Main() method shortly.

Essentially, you can compile a C# program into either an *assembly* or a *module*. If your program has one class that contains a Main() method, then it can be compiled directly into an assembly. This is a file that has a ".exe" filename extension. You can execute this file by typing its name at a command prompt or by double-clicking its icon. If your program has no Main() method, you will get the following error message when you compile:

`'[assembly file name]' does not contain a static 'Main' method suitable for an entry point`

A program with no Main() method can be compiled into a module using the csc compiler's `/target:module` argument. But even if a module contains executable code, it can't be loaded and run standalone by the virtual execution system if it doesn't have a Main() method. The VES doesn't know where to start execution.

Modules can be added to assemblies. In fact, modules of different languages that conform to the CLI specification can be combined with modules written in C# to form an executable assembly. Cross-language compatibility is one of the promises of both the .NET and the broader CLI initiative.

Structure Of A Simple Application

Example 6.1 gives the code for a simple C# application.

6.1 SimpleApp.cs

```
1    using System;
2
3    public class SimpleApp {
4       static void Main() {
5            Console.WriteLine("Howdy Stranger!");
6         }
7    }
```

Referring to Example 6.1 — SimpleApp is the name of the class that contains the Main() method. On line 1, a *using directive* signals the compiler that this source file refers to classes and constructs declared within the System namespace. (I cover namespaces in Chapter 9.) Specifically, in this short program I am using the System.Console class to get input from and send output to the command console.

Line 3 includes the keywords *public* and *class* to declare the class SimpleApp. At the end of line 3 there appears an opening curly brace '`{`'. This signals the beginning of SimpleApp's class *body*. Everything belonging to a class, that is all *fields*, *properties*, *methods*, etc., appear in the class body between the opening and closing curly braces.

The start of the Main() method begins on line 4. The keywords *static* and *void* are used to declare the Main() method. The Main() method, as you can see, contains an opening and closing parentheses "`( )`". The parentheses denote the beginning and ending of an optional *method parameter-list*. A *parameter* represents an object that will be passed to a method for processing when the method is called. (I'll talk more about Main() method parameters later.)

At the end of line 4, an opening curly brace denotes the beginning of the Main() method body. Any code appearing between the Main() method's opening and closing curly braces belongs to the Main() method. In the case of Example 6.1, the Main() method contains one line of code, line 5, which is a method call to the Console class's Write-

© 2008 Rick Miller — All Rights Reserved

Line() method. The WriteLine() method writes different types of objects to the console. In this case, I'm writing a string of characters (*i.e.,* a String object) to the console. Line 6 contains the Main() method's closing curly brace '}' and line 7 contains the SimpleApp class's closing curly brace.

To compile this program at the command prompt, you would save the source code in a file named SimpleApp.cs and use the csc compiler tool like so:

```
csc SimpleApp.cs
```

This creates an assembly named SimpleApp.exe. Figure 6-1 shows the results of running this program.

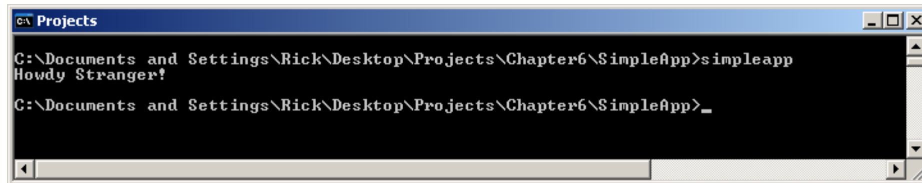


Figure 6-1: Results of Running Example 6.1

Purpose Of The Main() Method

The purpose of the Main() method is to provide an entry point for application execution. As I stated earlier, without a Main() method, the virtual execution system has no way of knowing where to start running a program.

Main() Method Signatures

The Main() method can have the following four *signatures*:

```
static void Main() { }
static void Main(string[] args) { }
static int Main() { }
static int Main(string[] args) { }
```

The term *method signature* refers to the combination of a method's name and its parameter list. A method's return type is *not* considered part of its signature, but the Main() method can optionally return an integer value, which yields four different versions. As these four method signatures show, the Main() method can return void (nothing) or optionally an integer value, and take either no parameters or a string array parameter.

The purpose of the string array parameter is to enable the passing of command-line arguments to the program. I will show you how to do this in Chapter 8 after you learn about arrays.

The SimpleApp class shown in Example 6.1 used the first version of Main(). It could have easily used the other versions as well. Example 6.2 shows the SimpleApp class employing the second version of the Main() method.

6.2 SimpleApp (Version 2)

```
1    using System;
2
3    public class SimpleApp {
4       static void Main(string[] args) {
5             Console.WriteLine("Howdy Stranger!");
6         }
7    }
```

Referring to Example 6.2 — this version of the SimpleApp class produces the same output when executed as that shown in Figure 6-1. In this case, the string array parameter named args is ignored.

Keep in mind that the only four versions of the Main() method authorized as entry points are those shown above. If you tried to use a method named Main() that took a different type or number of parameters, then you would receive a compiler warning. Let's see what happens if we try to use a different Main() method argument type. Example 6.3 gives the code.

6.3 SimpleApp (Version 3)

```
1    using System;
2
3    public class SimpleApp {
4       static void Main(int i) { // will not compile!
5             Console.WriteLine("Howdy Stranger!");
6         }
7    }
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 6.3 — the Main() method string parameter has been replaced with an integer parameter. When you compile this version of the program, it produces the error messages shown in Figure 6-2.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\SimpleApp_V3>csc simpleapp.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

SimpleApp.cs(8,17): warning CS0028: 'SimpleApp.Main(int)' has the wrong signature to be an entry point
error CS5001: Program 'c:\Documents and Settings\Rick\Desktop\Projects\Chapter6\SimpleApp_V3\SimpleApp.exe' does not
        contain a static 'Main' method suitable for an entry point

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\SimpleApp_V3>
```

Figure 6-2: Results of Compiling Example 6.3 with Improper Main() Method Signature

Quick Review

A simple C# application is a class that contains a Main() method. The purpose of the Main() method is to provide an entry point for program execution. There are four authorized Main() method signatures. A class that contains a Main() method can be compiled into an executable assembly. A class with no Main() method can be compiled into a module. Modules can be added to assemblies. Modules created in CLI compliant languages other than C# can be compiled with C# modules to form executable assemblies.

Identifiers And Reserved Keywords

Table 6-1 lists C# language reserved keywords.

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

Table 6-1: C# Reserved Keywords

© 2008 Rick Miller — All Rights Reserved

Referring to Table 6-1 — there's no need to memorize the entire list. In time, as you write increasingly complex programs, you will come to know most of them intimately. The important thing to note right now is that reserved keywords have special meaning in the C# language. You can't hijack them for your own purpose.

In the SimpleApp code shown in Example 6.1, you saw several keywords put to use. These included *class*, *public*, *static*, *void*, *string*, and *using*. The *class* keyword is used to introduce a new class type name, in this case the string of characters "SimpleApp". The string of characters "SimpleApp" is known as an *identifier*. The act of programming requires you to invent names for lots of things in your programs like variables, constants, class and method names. So long as the names you choose for these objects are different from the reserved keywords, you'll be fine. But what would happen if you were to try and introduce a new name for an object within your program that has already been reserved? Let's see what happens. Example 6.4 gives the code for a naughty little program that tries to declare a class named "class".

6.4 Naughty Program

```
1    using System;
2
3    public class class { // <-- will cause an error when compiled
4      static void Main(){
5        Console.WriteLine("Bad, bad program...!");
6      }
7    }
```

Referring to Example 6.4 — on line 3 an attempt is made to introduce a new class named "class". But since class is a reserved keyword, this causes the compiler to pitch quite a fit, as is shown in Figure 6-3.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\NaughtyProgram>csc class.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

class.cs(3,14): error CS1041: Identifier expected, 'class' is a keyword
class.cs(3,14): error CS1513: } expected
class.cs(3,20): error CS1001: Identifier expected
class.cs(3,14): error CS0101: The namespace '<global namespace>' already contains a definition for '?'
class.cs(3,8): (Location of symbol related to previous error)

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\NaughtyProgram>
```

Figure 6-3: Errors Produced when Attempting to Reintroduce a Reserved Keyword

Identifier Naming Rules

It's easy to avoid trouble in formulating identifier names if you take the time to give the objects in your programs names that make sense within the context of the problem at hand. Creating valid identifiers is easy, as you'll see. What takes a little more skill is effectively naming objects within a program that correspond to real world objects in the problem domain. For more information on this topic, see the discussion on isomorphic mapping in Chapter 1.

Identifiers can start with a letter or the underscore '`_`' character. The starting letter can be uppercase or lowercase. The starting character can be followed by any number of letters, underscores and decimal digits. Unicode character escape sequences can be used as well, but putting these in your identifiers makes them difficult to read and understand.

Although I said earlier that reserved keywords cannot be used as identifiers, I will recant somewhat and say that if you add the '`@`' character in front of a keyword, you can use it as an identifier. Example 6.5 shows how this is done.

6.5 Somewhat Bad Program

```
1    using System;
2
3    public class @class { // <-- This will work...!
4      static void Main(){
5        Console.WriteLine("Works but not recommended...!");
6      }
7    }
```

Referring to Example 6.5 — the '`@`' character is added to the beginning of the second occurrence of the class keyword on line 3, which now forms a valid identifier. (The identifier is "`@class`".) Although this works, I don't recommend doing this as the inevitable result will be code that's hard to read, understand, and maintain. I leave it up to you to compile Example 6.5 and see for yourself the results of its execution.

 © 2008 Rick Miller — All Rights Reserved

Quick Review

Identifiers are sequences of characters that represent names of objects in a program. Identifiers are used to formulate the name of classes, structures, methods, variables, constants, properties, fields, enums, etc.

Identifiers can start with either an uppercase or lowercase letter or an underscore '_' character followed by any number of letters, digits, and underscores.

Reserved keywords are identifiers that have special meaning within the C# language. You cannot reintroduce a reserved keyword as a name for an object within your program. You can, however, prepend the '@' symbol to a reserved keyword to formulate a valid identifier, but I discourage you from doing this as it renders code hard to read, understand, and maintain.

Types

C# is a strongly typed programming language. The term strongly typed means that all objects in a C# program must be associated with a particular type. An object's type is a specification of the legal operations that can be performed on that object. For example, the '+' operator can be applied to integer (int) objects, and the Append() method can be called on StringBuilder objects. Generally speaking, if you try to perform an operation on an object that its particular type does not allow, you will get a compiler error.

There are two categories of types in the C# language: *value types* and *reference types*. Figure 6-4 gives the complete C# type hierarchy.

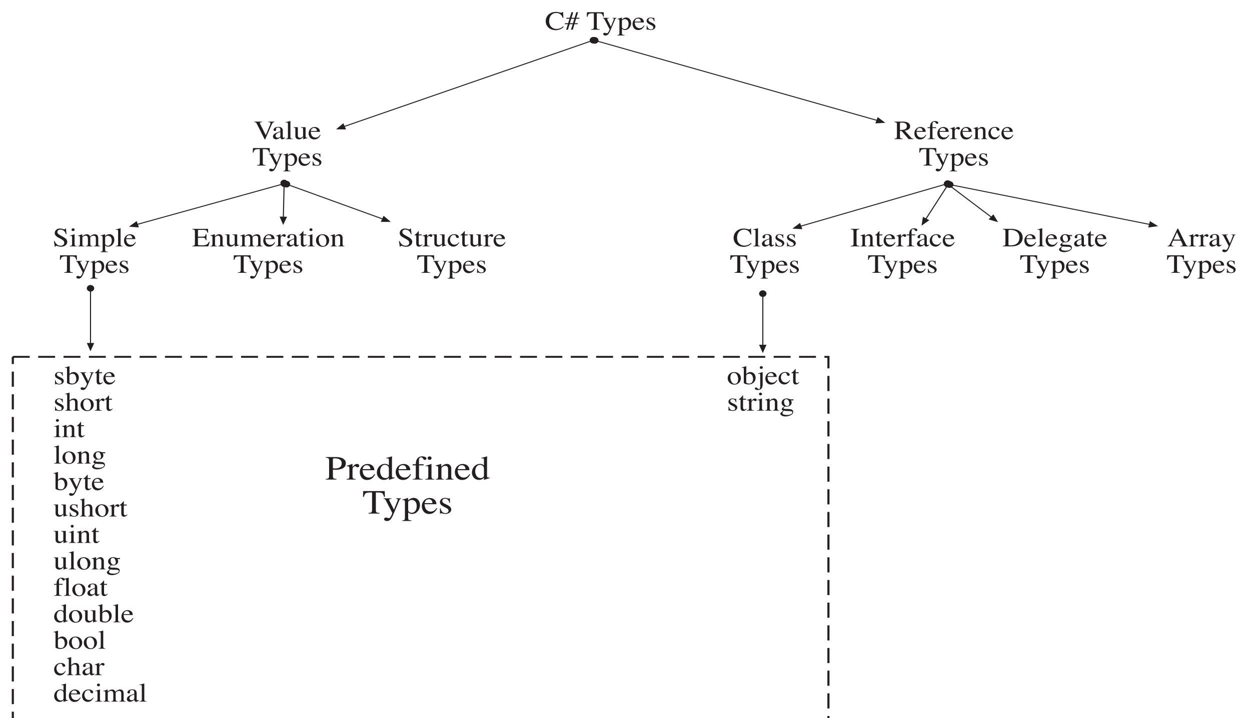


Figure 6-4: C# Type Hierarchy

Referring to Figure 6-4 — value types and reference types each have several type subcategories. The important thing to note in Figure 6-4 is the existence of the C# predefined types. These are the types that are built into the language. Notice that the predefined type names all start with lowercase letters. All but two of the predefined types are simple value types. The types *object* and *string* are class types, which is a subcategory of reference types. Value types behave differently from reference types. I explain these behavioral differences in the next section.

© 2008 Rick Miller — All Rights Reserved

Value Type Variables vs. Reference Type Variables

This section explains the differences between value type and reference type variables.

Value Type Variables

A value type variable contains its very own copy of its data. Let's take a look at a simple example of value types in action.

6.6 ValueTypeTest.cs

```
1    using System;
2
3    public class ValueTypeTest {
4      static void Main(){
5        int i = 0;
6        int j = i;
7        j = j+1;
8        Console.WriteLine("The value of i is: " + i);
9        Console.WriteLine("The value of j is: " + j);
10     }
11   }
```

Referring to Example 6.6 — An integer value type variable named i is declared and initialized on line 5. The term *variable* means a named storage location in memory whose value can be changed during program execution. On line 6, another integer variable named j is declared and initialized to the value of i. On line 7, a simple addition operation is performed on the variable j adding 1 to its value. Adding 1 to the variable j does not affect the value of the variable i. The code on lines 8 and 9 print the values of i and j to the console. Figure 6-5 shows the results of running this program.

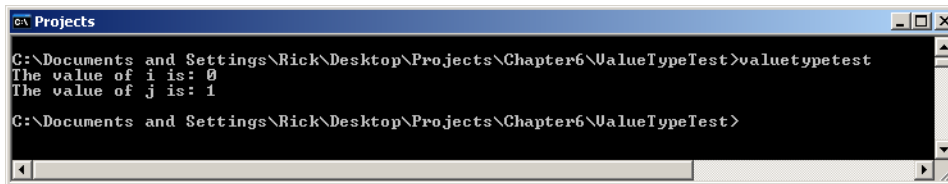


Figure 6-5: Results of Running Example 6.6

Reference Type Variables

A reference type variable contains the memory address of a reference type object. Two different reference type variables can point to the same reference type object in memory. The following program offers an example.

6.7 ReferenceTypeTest.cs

```
1    using System;
2    using System.Text;
3
4    public class ReferenceTypeTest {
5      static void Main(){
6        StringBuilder sb1 = new StringBuilder();
7        StringBuilder sb2 = sb1;
8        sb1.Append("Howdy Pawdner!");
9        Console.WriteLine(sb1);
10       Console.WriteLine(sb2);
11     }
12   }
```

Referring to Example 6.7 — on line 2, another using directive provides shortcut name access to the StringBuilder class located in the System.Text namespace. On lines 6 and 7, in the body of the Main() method, two StringBuilder reference variables named sb1 and sb2 are declared and initialized. Notice that in order to create a StringBuilder object, you must use the *new* operator as is shown on line 6. On line 7, the StringBuilder variable named sb2 is initialized to the same value as sb1. Remember, reference type variables store memory addresses to objects located in memory. So, when the value of sb1 is assigned to sb2, sb2 is being assigned a memory address. Now sb1 and sb2 both "point" to or "reference" the same StringBuilder object in memory. Any operation performed on the object pointed to by sb1 affects the object pointed to by sb2 since, in this case, it is the same object. This is what happens when the Append() method is called via the sb1 variable adding the character string "Howdy Pawdner!". Note the results of running this program shown in Figure 6-6.

 © 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\ReferenceTypeTest>referencetypetest
Howdy Pawdner!
Howdy Pawdner!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\ReferenceTypeTest>_
```

Figure 6-6: The Results of Running Example 6.7

Maybe Some Pictures Will Help

Figure 6-7 offers a simple conceptual view of value-type memory allocation based on the code presented in Example 6.6.

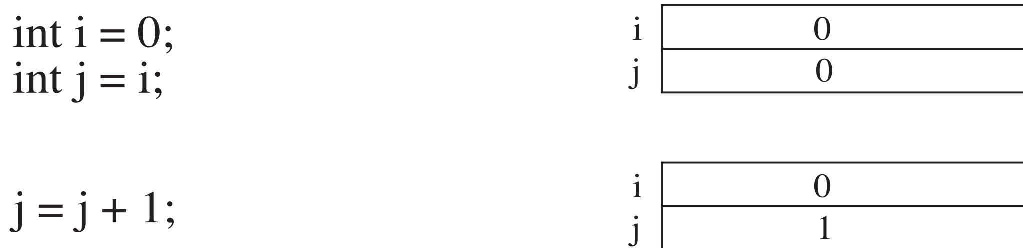


Figure 6-7: Value Type Memory Allocation

Referring to Figure 6-7 — the integer variables i and j each hold their very own copy of their assigned values. When the value of variable i is assigned to the variable j, a copy of i's value, in this case 0, is made and stored in j's memory location.

Figure 6-8 shows a simple conceptual view of reference type memory allocation based on the code presented in Example 6.7.

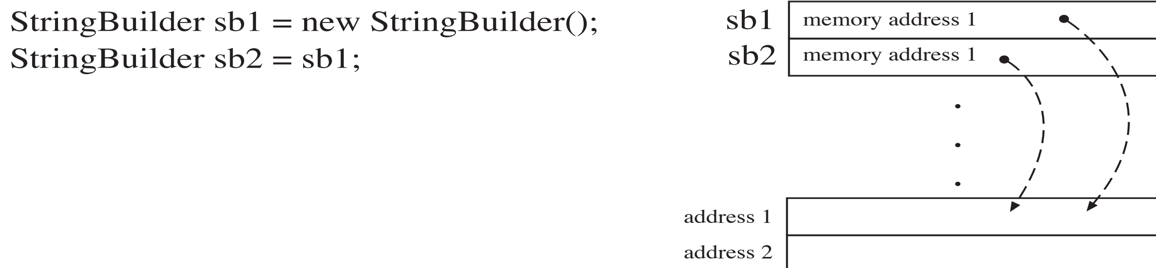


Figure 6-8: Reference Type Memory Allocation

Referring to Figure 6-8 — the StringBuilder variable sb1 contains the memory address of a StringBuilder object. The StringBuilder object was created with the expression "new StringBuilder()" which creates the object and returns the address of the object's location in memory. When the value of sb1 is assigned to the variable sb2, both variables will point to the same object in memory. When two reference variables point to the same object, any operation performed on one affects the other, as is shown in Figure 6-9.

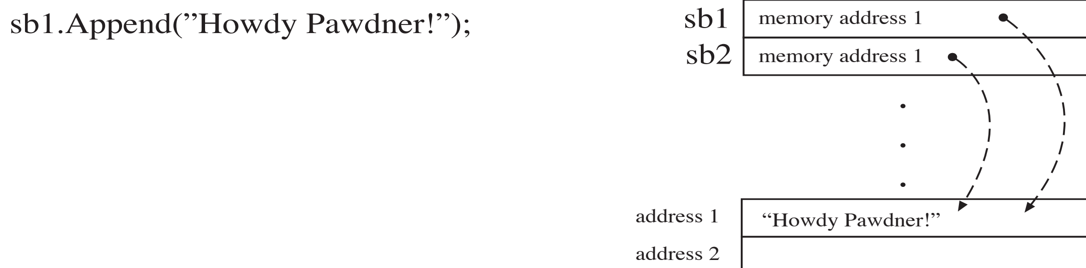


Figure 6-9: Results of Calling the Append() Method via the sb1 Variable

© 2008 Rick Miller — All Rights Reserved

Mapping Predefined Types To System Structures

All the predefined types correspond to structures within the System namespace of the .NET API. For example, the predefined simple type *int* is mapped to the *System.Int32* structure. The System.Int32 structure inherits from the *System.ValueType* class, as do all value types and enumerations. Example 6.8 gives an alternative version of the ValueTypeTest code originally presented in Example 6.6.

6.8 ValueTypeTest.cs (Version 2)

```
1    using System;
2
3    public class ValueTypeTest {
4      static void Main(){
5        Int32 i = 0;
6        Int32 j = i;
7        j = j+1;
8        Console.WriteLine("The value of i is: " + i);
9        Console.WriteLine("The value of j is: " + j);
10     }
11   }
```

Referring to Example 6.8 — compare this program with Example 6.6. Notice the only difference between the two programs is the substitution here of the type Int32 for the simple type int. Table 6-2 lists the predefined types along with their corresponding System namespace structures, default values, and value ranges.

Type	Description	System Namespace Structure or Class	Default Value† / Value Range
object	The base class of all types	Object Class	Default value: null
string	A sequence of Unicode code units	String Class	Default value: null
sbyte	8-bit signed integral type	SByte Structure	Default value: 0 -128 to 127
short	16-bit signed integral type	Int16 Structure	Default value: 0 -32768 to 32767
int	32-bit signed integral type	Int32 Structure	Default value: 0 -2,147,483,648 to 2, 147,483,647
long	64-bit signed integral type	Int64 Structure	Default value: 0 -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
byte	8-bit unsigned integral type	Byte Structure	Default value: 0 0 to 255
ushort	16-bit unsigned integral type	UInt16 Structure	Default value: 0 0 to 65535
uint	32-bit unsigned integral type	UInt32 Structure	Default value: 0 0 to 4,294,967,295
ulong	64-bit unsigned integral type	UInt64 Structure	Default value: 0 0 to 18,446,744,073,709,551,615
float	single-precision floating point type	Single Structure	Default value: 0.0 -3.402823e38 to 3.402823e38
double	double-precision 64-bit floating point type	Double Structure	Default value: 0.0 -1.79769313486232e308 to 1.79769313486232e308

Table 6-2: Predefined Type Mappings, Default Values, and Value Ranges

© 2008 Rick Miller — All Rights Reserved

Type	Description	System Namespace Structure or Class	Default Value† / Value Range
bool	Represents true or false	Boolean Structure	Default value: false true or false
char	character type (Unicode code unit)	Char Structure	Default value: \u0000 Any Unicode value
decimal	decimal type with at least 28 significant digits	Decimal Structure	Default value: 0 -79,228,162,514,264,337,593,543,950,335 to 79,228,162,514,264,337,593,543,950,335
† Default values are assigned to class or structure fields. Local method variables must be explicitly assigned.			

Table 6-2: Predefined Type Mappings, Default Values, and Value Ranges

Quick Review

C# has two kinds of types: value types and reference types. Value type variables contain the actual data as defined by the type. Reference type variables contain a reference to an object in memory. Two or more reference type variables can reference the same object in memory. The C# predefined types map to structures within the System namespace. System.Object is the base type for all types.

Statements, Expressions, and Operators

Statements are the fundamental building blocks of C# programs. A statement can be thought of as the smallest standalone element of a program, and programs are built using sequences of statements. The simplest type of statement is the *empty statement*. An empty statement would look like this:

```
;
```

It's just a lonely semicolon on a line by itself, although it doesn't have to be on a line by itself.

You've already seen statements in action in this chapter's example programs. The following line of code is taken from Example 6.6:

```
int i = 0;
```

This is an example of a *local variable* declaration statement. It's a local variable declaration because this line of code appeared within the body of a method, in this case, the Main() method. This variable declaration statement contains within it an expression statement. The assignment operator '=' assigns the value 0 to the variable i. Complex statements can be formed by combining statements within statements.

Notice that the statement above is terminated by the semicolon ';' character. The semicolon character indicates a line of execution. Note the following three lines of code:

```
1    int i = 0;
2    int j = i;
3    int j = j + 1;
```

The results of the execution of line 1 will be fully complete before line 2 begins execution. And again, the results of line 2 will be fully available when line 3 begins execution.

Statement Types

There are nineteen different types of statements in the C# language. These are listed in Table 6-3.

© 2008 Rick Miller — All Rights Reserved

Statement	Statement Lists and Block Statements
	Labeled Statements and goto Statements
	Local Constant Declarations
	Local Variable Declarations
	Expression Statements
	if Statements
	switch Statements
	while Statements
	do Statements
	for Statements
	foreach Statements
	break Statements
	continue Statements
	return Statements
	yield Statements
	throw Statements and try Statements
	checked and unchecked Statements
	lock Statements
	using Statements

Table 6-3: C# Statement Types

Referring to Table 6-3 — a statement can be any of the statement types listed in the right column. All of these statement types are discussed throughout the book, so I will not give examples of each here. With sufficient programming experience, their use becomes second nature. I will, however, elaborate on how the different types of C# operators are used in expression statements. This is the topic of the next section.

Operators And Their Use

Table 6-4 lists the C# operators by expression category and precedence.

Expression Category	**Operators**
Primary	x.y f(x) a[x] x++ x-- new typeof checked unchecked
Unary	+ - ! ~ ++x --x (T)x
Multiplicative	* / %
Additive	+ -

Table 6-4: Operator Categories by Precedence

© 2008 Rick Miller — All Rights Reserved

Expression Category	Operators
Shift	`<< >>`
Relational and Type-Testing	`< > <= >= is as`
Equality	`== !=`
Logical AND	`&`
Logical XOR	`^`
Logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Conditional	`?:`
Assignment	`= *= /= %= += -= <<= >>= &= ^= |=`

Table 6-4: Operator Categories by Precedence

Operator Precedence and Associativity

The term *operator precedence* refers to the order in which the C# compiler evaluates the operators in an expression statement. Consider for a moment the following line of code:

```
int i = 25 - 2 * 2;
```

The variable i is being assigned some value. But what value? If you leave it to the compiler to apply its precedence rules, the variable i will be assigned the value 21. The multiplication operator '`*`' has a higher precedence than the subtraction operator '`-`'. This may or may not be the way you want the expression to evaluate.

The term *associativity* refers to the direction in which the C# compiler performs a series of operations. Binary arithmetic operators like the multiplication and subtraction operators used above have left-to-right associativity. For example, given an expression of the form `2-2-2` the order in which the subtraction operations are performed is `(2-2)-2`. In the example statement given above, the expression `2*2` was performed first because the multiplication operator has a higher precedence than the subtraction operator.

Assignment operations have right-to-left associativity. Thus, the compiler evaluates an expression of the form `i=j=k` as `i=(j=k)`.

Forcing Operator Precedence and Associativity Order With Parentheses

You can force the compiler to evaluate a complex expression a particular way by using parentheses. If we apply parentheses to the expression shown above in the following manner:

```
int i = (25 - 2) * 2;
```

This will cause the subtraction operator '`-`' to be evaluated before the multiplication operator '`*`', yielding the value 46. It's good programming practice to always use parentheses to show how you intend an expression to be evaluated. Doing so eliminates the possibility of making hard-to-find mistakes and makes your code easier to read and understand.

Operators and Operands

Operators are applied to operands. For example, in the following expression fragment:

```
25 - 2
```

The subtraction operator takes two operands. In the following code fragment:

```
i = 25 - 2
```


© 2008 Rick Miller — All Rights Reserved

The subtraction operation with its two operands is evaluated first, yielding a value of 23. This leaves two operands for the assignment operator '=' to work on: i and 23. As you will soon see, some operators operate on one operand, some on two operands, and one on three operands.

Operator Usage Examples

In this section, I demonstrate the use of one or more operators from each of the operator categories listed in Table 6-4. You will most assuredly encounter all of these operators in more depth as you progress through the book.

Primary Expression Operators

Primary expression operators have the highest precedence. The use of parentheses with these is not usually necessary, nor legal in some cases, to force an unnatural association. You've seen several primary expression operators in action already in this chapter. These included the `new` operator and the member access '`.`' operator.

The new operator creates a reference type object. The member access operator is used to access object members. Consider, for example, the following two lines of code:

```
StringBuilder sb1 = new StringBuilder();
sb1.Append("Adding this string to the sb1 object.");
```

The `new` operator creates a new StringBuilder object in memory. The assignment operator assigns the resulting memory address to the StringBuilder reference variable sb1. The StringBuilder's Append() method is called via the sb1 variable with the help of the member access operator.

Two other primary operators you will frequently use are the postfix increment and decrement operators, '`++`' and '`--`' respectively. Example 6.9 shows the operators in use.

6.9 PrimaryOperatorTest.cs

```
1    using System;
2
3    public class PrimaryOperatorTest {
4      static void Main(){
5        int i = 0;
6        Console.WriteLine(i++); // writes value of i then increments
7        Console.WriteLine(i--); // writes value of i then decrements
8        Console.WriteLine(i);   // simply writes value of i
9      }
10   }
```

Referring to Example 6.9 — an integer variable named i is created on line 5. On lines 6 and 7 the increment and decrement operators are applied to the variable i, which is being used as an argument to the Console.WriteLine() method. Notice on line 6 that the increment operator appears to the right of i. This is the postfix application of this operator, which means "increment the value of i after the statement has been evaluated." The effects here are that the value 0 is written to the console.

On line 7, the decrement operator appears to the right of i. The effect is that the current value of i, which is now 1, is printed to the console and then decremented. Line 8 simply prints the last value of i to the screen. Figure 6-10 shows the results of running this program.

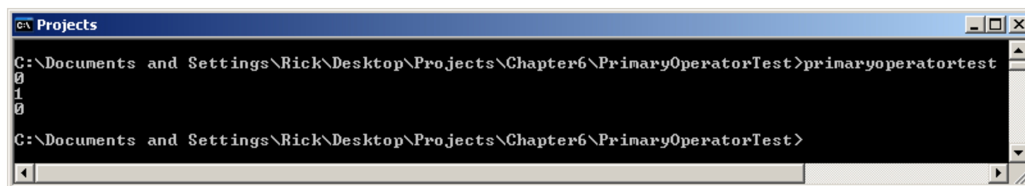


Figure 6-10: Results of Running Example 6.9

Unary Expression Operators

Unary expression operators operate on one operand. The unary expression operators include the prefix increment '`++`' and decrement '`--`', the plus '`+`' and minus '`-`', the logical negation '`!`', the bitwise complement '`~`', and the cast '$(T_2)T_1$'. The cast operator forces a change from one type T_1 to another type T_2. The plus and minus unary operators change the sign of integral and floating point numbers. The logical negation operator changes the value of

 © 2008 Rick Miller — All Rights Reserved

boolean expressions from false to true and vice versa. The bitwise complement operator switches the bit values of unsigned integral types. (*i.e.,* If a bit is set to 1 it will be changed to 0.)

Example 6.10 offers a short program showing some of these operators in action.

6.10 UnaryOperatorTest.cs

```
1     using System;
2
3     public class UnaryOperatorTest {
4       static void Main(){
5         int i = 25;
6         bool bool_var = true;
7         uint j = 1;
8         Console.WriteLine(-i);
9         Console.WriteLine(!bool_var);
10        Console.WriteLine(~j);
11      }
12    }
```

Referring to Example 6.10 — on line 5, an integer variable named i is declared and initialized to the value 25. On line 6, a boolean variable named bool_var is declared and initialized to the value true. On line 7, an unsigned integer (uint) variable named j is declared and initialized to the value 1. Each of these variables is then printed to the console after its value has been affected by the various unary operators. Figure 6-11 gives the results of running this program.

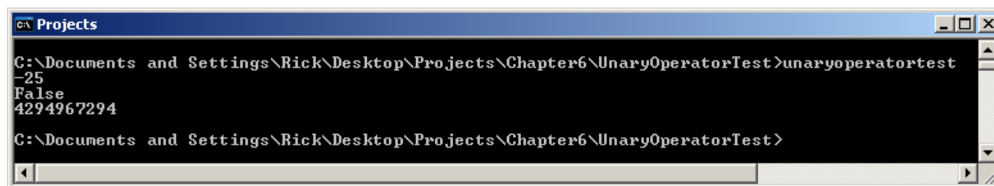


Figure 6-11: Results of Running Example 6.10

Multiplicative Expression Operators

The multiplicative expression operators include the multiplication '*', division '/', and the remainder '%' operators. The remainder operator '%' is also referred to as the *modulus* operator.

The multiplicative operators are binary operators in that they operate on two operands. You are already familiar with the notion of how the multiplication and division operators work from your elementary arithmetic background. What you need to be acutely aware of, however, is how each of these operators behaves given different types of numbers. For example, what happens if you multiply two numbers and try to assign the result into a variable type that's too small to accommodate the resultant value? What happens if you divide two integer values vs. two floating point values? If you always keep in mind the relative range of values the different simple types can represent, you will avoid most problems. Operations that attempt to assign a large value to a type that's too small to represent it will result in both a loss of precision and in a compiler warning.

The remainder operator performs a division operation on integral values and returns only the remainder. Example 6.11 shows the remainder operator in action.

6.11 RemainderOperatorTest.cs

```
1     using System;
2
3     public class RemainderOperatorTest {
4       static void Main(){
5         int i = 10;
6         int j = 5;
7         int k = 3;
8         Console.WriteLine(i%j);
9         Console.WriteLine(i%k);
10      }
11    }
```

Referring to Example 6.11 — three integer variables i, j, and k are declared and initialized to the values 10, 5, and 3, respectively. Line 8 prints out the result of the remainder operator applied to the variables i and j. Line 9 prints out the result of the remainder operator applied to the variables i and k. Figure 6-12 shows the results of running this program.

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\RemainderOpTest>remainderoperatortest
0
1

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\RemainderOpTest>
```

Figure 6-12: Results of Running Example 6.11

Additive Expression Operators

The additive expression operators include the arithmetic addition '+' and subtraction '–' operators. I will forego an example of these operators as they are easy and intuitive to use.

Shift Expression Operators

The shift expression operators include the left shift '<<' and right shift '>>' operators. The shift operators perform bit shifting operations.

The important thing to know about the bit shifting operators is how they behave when applied to different integral types. If the value being shifted is a signed integral type, then an arithmetic shift is performed. An arithmetic shift means that the sign of the value is preserved as the bits are shifted right. If the value being shifted is an unsigned integral type, a logical shift occurs and high-order empty bit positions are set to zero. Let's take a look at the shift operators in action in Example 6.12.

6.12 ShiftOperatorTest.cs

```
1    using System;
2
3    public class ShiftOperatorTest {
4      static void Main(){
5        short i =  -0x000F;
6        short j =   0x000F;
7        Console.WriteLine("The value of i before the shift: " + i);
8        Console.WriteLine("The value of j before the shift: " + j);
9        Console.WriteLine("The value of i after the shift: " + (i >> 2));
10       Console.WriteLine("The value of j after the shift: " + (j >> 2));
11     }
12   }
```

Referring to Example 6.12 — two short variables named i and j are declared and initialized using *hexadecimal literal* values representing -15 and 15 respectively. Lines 7 and 8 print these values of i and j to the console. Lines 9 and 10 print the values of i and j after the right shift operator has been applied, shifting the bits two places to the right. What do you think the new values will be? Figure 6-13 shows the results of running this program. Cover the figure and try to work it out before proceeding. A detailed explanation follows the figure.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\ShiftOpTest>shiftoperatortest
The value of i before the shift: -15
The value of j before the shift: 15
The value of i after the shift: -4
The value of j after the shift: 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\ShiftOpTest>
```

Figure 6-13: Results of Running Example 6.12

Referring to Figure 6-13 — after the shift, the value of i is -4 and the value of j is 3. Here's a brief explanation as to why they are different. The value 15 is represented in hexadecimal as the letter F. The hexadecimal value F is represented in binary as 1111. A short type is sixteen digits long, therefore the full binary for the positive number 15 is:

0000000000001111

The value of the variable i is -15. To convert the binary value 15 to -15, you need to invert the bits and add 1. This is known as 1's complement. The resulting binary value representing the number -15 looks like this:

1111111111110001

When this string of binary digits is shifted two places to the right, the new value becomes:

© 2008 Rick Miller — All Rights Reserved

```
1111111111111100
```

This is the binary representation of the decimal value -4.

The value of the variable j is also shifted to the right two places, but because it's a positive value, the left-most binary digits are replaced with 0. The binary value of j after the shift looks like this:

```
0000000000000011
```

This is the binary representation for the decimal value 3.

Relational, Type-Testing, and Equality Expression Operators

This category of operators includes the comparison operators equals '`==`', not equals '`!=`', less than '`<`', greater than '`>`', less than or equal to '`<=`'. and greater than or equal to '`>=`'. It also includes the type testing operators '`is`' and '`as`'.

The comparison operators work on integral, floating point, decimal, and enumeration types, The '`==`' and '`!=`' operators work on boolean, reference, string, and delegate types. The behavior of these operators is summed up in Table 6-5.

Operator	Behavior	Operands
`<`	Returns true if left operand is less than the right operand; false otherwise	numeric types enumeration types reference types if overloaded
`>`	Returns true is left operand is greater than the right operand; false otherwise	numeric types enumeration types reference types if overloaded
`<=`	Returns true if the left operand is less than or equal to the right operand; false otherwise	numeric types enumeration types reference types if overloaded
`>=`	Returns true if the left operand is greater than or equal to the right operand; false otherwise	numeric types enumeration types reference types if overloaded
`==`	Returns true if the left operand is equal to the right operand; false otherwise	numeric types enumeration types boolean values string objects delegate types reference types if overloaded
`!=`	Returns true if the left operand is not equal to the right operand; false otherwise	numeric types enumeration types boolean values string objects delegate types reference types if overloaded

Table 6-5: Comparison Operator Behavior

The behavior of these operators is easy to understand in the context of numbers. However, the behavior of the '`==`' and '`!=`' operators and how they work for reference objects begs for an example. These two operators will work on string objects as expected but only because the String class provides a definition for them. In other words, string objects know how to behave when compared to each other with the '`==`' and '`!=`' operators.

User-defined classes that do not overload the '`==`' or '`!=`' operators will be compared to each other according to the rules the operators follow when comparing ordinary objects. Let's look at an example. Example 6.13 gives the code.

© 2008 Rick Miller — All Rights Reserved

6.13 ReferenceEqualityTest.cs

```
1    using System;
2
3    public class ReferenceEqualityTest {
4      static void Main(){
5         Object o1 = new Object();
6         Object o2 = new Object();
7         Object o3 = o2;
8         String s1 = "Hello";
9         String s2 = "Hello";
10        String s3 = "World";
11        Console.WriteLine(o1 == o2);
12        Console.WriteLine(o1 != o2);
13        Console.WriteLine(o2 == o3);
14        Console.WriteLine(s1 == s2);
15        Console.WriteLine(s1 == s3);
16     }
17   }
```

Referring to Example 6.13 — three Object reference variables named o1 through o3 are declared and initialized on lines 5 through 7. The variables o1 and o2 point to unique objects. The variable o3 is assigned the same address as the variable o2. This means that the variables o2 and o3 now point to the same object.

On lines 8 through 10, three String variables are created. The variables s1 and s2 each point to identical string values “Hello”. The variable s3 points to a string whose value is “World”. Now study the results of running this program as shown in Figure 6-14.

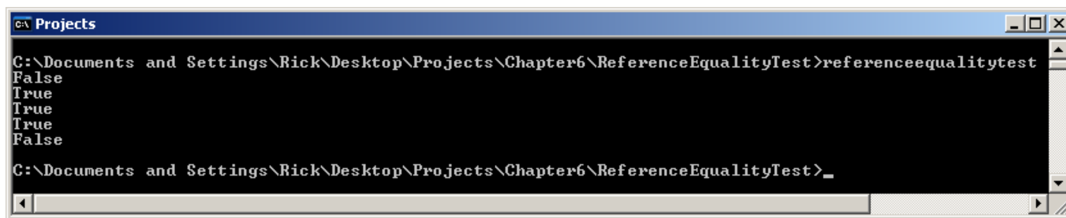


Figure 6-14: Results of Running Example 6.13

Referring to Figure 6-14 — on line 11, the expression o1 == o3 evaluates to false because the variables o1 and o2 point to different objects. The “==” operator’s natural behavior as defined in the Object class is to test if we are comparing the same object. If not, the operator returns false. On the next line, the expression o1 != o2 returns true as expected. On line 3, the expression o3 == o2 returns true because the variables o2 and o3 do in fact point to the same object and so they must be equal.

These same operators behave somewhat differently when used with String objects. Notice now on line 14 that if two different Strings are compared the result will be true if their values (*i.e.*, the characters they contain) are identical. Different String objects with different values will return false, as is shown on the last line.

Logical AND, OR, and XOR Expression Operators

The logical AND ‘`&`’, OR ‘`|`’, and XOR ‘`^`’ operators behave differently according to their parameter types. Table 6-6 summarizes the logical operator behavior.

Operator	Behavior	Operands
`&`	Integral operands: x & y performs bitwise logical AND Enumeration operands: x & y performs bitwise logical AND Boolean operands: Performs conditional AND comparison	int, uint, long, ulong enumeration boolean
`|`	Integral operands: x	y performs bitwise logical OR Enumeration operands: x	y performs bitwise logical OR Boolean operands: Performs conditional OR comparison	int, uint, long, ulong enumeration boolean

Table 6-6: Logical Operator Behavior

© 2008 Rick Miller — All Rights Reserved

Operator	Behavior	Operands
^	Integral operands: x ^ y performs bitwise logical XOR Enumeration operands: x ^ y performs bitwise logical XOR Boolean operands: Performs conditional XOR comparison	int, uint, long, ulong enumeration boolean

Table 6-6: Logical Operator Behavior

Logical Operations on Integral Operands

When presented with integral operands, the logical operators perform bitwise logical operations on their operands according to the truth tables shown in Figure 6-15. Example 6.14 shows these operators in action.

6.14 LogicalOperatorTest.cs

```
1    using System;
2
3    public class LogicalOperatorTest {
4      static void Main(){
5        int i = 0xFFFF;
6        int mask_1 = 0x0000;
7        int mask_2 = 0x0003;
8        int mask_3 = 0xFFFF;
9        Console.WriteLine("FFFF & 0000 = " + (i & mask_1));
10       Console.WriteLine("FFFF | 0000 = " + (i | mask_1));
11       Console.WriteLine("FFFF & 0003 = " + (i & mask_2));
12       Console.WriteLine("FFFF | 0003 = " + (i | mask_2));
13       Console.WriteLine("FFFF ^ FFFF = " + (i ^ mask_3));
14     }
15   }
```

X	Y	X & Y
0	0	0
1	0	0
0	1	0
1	1	1

X	Y	X | Y
0	0	0
1	0	1
0	1	1
1	1	1

X	Y	X ^ Y
0	0	0
1	0	1
0	1	1
1	1	0

Figure 6-15: Logical AND, OR, and XOR Truth Tables

Referring to Example 6.14 — on line 5, an integer variable i is declared and initialized to the hexadecimal value FFFF. On lines 6 through 8, three more integer variables named mask_1 through mask_3 are declared and initialized with the hexadecimal values 0000, 0003, and FFFF, respectively. Lines 9 through 13 use the logical operators to perform bit manipulation operations on the variable i using the various mask values. Figure 6-16 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\LogicalOperatorTest>LogicalOperatorTest
FFFF & 0000 = 0
FFFF | 0000 = 65535
FFFF & 0003 = 3
FFFF | 0003 = 65535
FFFF ^ FFFF = 0

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\LogicalOperatorTest>
```

Figure 6-16: Results of Running Example 6.14

Logical Operations on Enumeration Operands

The logical operators work with enumeration type operands. An enumeration is a type that represents any one of a set of authorized values. Enumeration types are declared with the `enum` keyword and can be defined outside of or within the body of a class, as the code in Example 6.15 illustrates.

 © 2008 Rick Miller — All Rights Reserved

6.15 LogicalOperationEnumTest.cs

```
1    using System;
2
3    public class LogicalOperatorEnumTest {
4
5      public enum EYE_COLOR {BLACK, BROWN, HAZEL, BLUE, GREY};
6
7      static void Main(){
8        Console.WriteLine(EYE_COLOR.BLACK & EYE_COLOR.BROWN);
9        Console.WriteLine(EYE_COLOR.BROWN & EYE_COLOR.BROWN);
10       Console.WriteLine(EYE_COLOR.BLACK & EYE_COLOR.BLUE);
11       Console.WriteLine(EYE_COLOR.BLACK | EYE_COLOR.BROWN);
12       Console.WriteLine(EYE_COLOR.BROWN | EYE_COLOR.HAZEL);
13       Console.WriteLine(EYE_COLOR.BLACK | EYE_COLOR.BLUE);
14       Console.WriteLine(EYE_COLOR.BLACK ^ EYE_COLOR.BROWN);
15       Console.WriteLine(EYE_COLOR.BROWN ^ EYE_COLOR.BROWN);
16       Console.WriteLine(EYE_COLOR.BLACK ^ EYE_COLOR.BLUE);
17     }
18   }
```

Referring to Example 6.15 — an enumerated type named EYE_COLOR is declared on line 5, and within the curly braces there appear five names: BLACK, BROWN, HAZEL, BLUE, and GREY. The enumeration value BLACK equates to the value 0, which is the default value for the first enumeration value unless explicitly set to be something else. The next enumeration value BROWN is assigned the value 1, and so on. (Enumerated types are covered in more detail in Chapter 9.)

Essentially, the logical operators treat enumeration types like they treat integers, which they ultimately are. Figure 6-17 shows the results of running Example 6.15.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\LogicalOperatorTest>logicaloperatorenumtest
BLACK
BROWN
BLACK
BROWN
BLUE
BLUE
BROWN
BLACK
BLUE

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\LogicalOperatorTest>
```

Figure 6-17: Results of Running Example 6.15

Logical Operations on Boolean Operands

The logical operators also operate on operands of type boolean. A boolean argument can be a boolean literal, a boolean variable, or a conditional expression that evaluates to a boolean value. Example 6.16 demonstrates the use of the logical operators on boolean literals. Just keep in mind that where the keywords “true” or “false” appear in the program a complex expression that evaluates to “true” or “false” could be substituted. Figure 6-18 gives the results of running this program. Compare its output with the truth tables given in Figure 6-15.

6.16 LogicalBoolTest.cs

```
1    using System;
2
3    public class LogicalBoolTest {
4       static void Main(){
5         Console.WriteLine("true  & true  = " + (true  & true));
6         Console.WriteLine("true  & false = " + (true  & false));
7         Console.WriteLine("false & true  = " + (false & true));
8         Console.WriteLine("false & false = " + (false & false));
9         Console.WriteLine("------------");
10        Console.WriteLine("true  | true  = " + (true  | true));
11        Console.WriteLine("true  | false = " + (true  | false));
12        Console.WriteLine("false | true  = " + (false | true));
13        Console.WriteLine("false | false = " + (false | false));
14        Console.WriteLine("------------");
15        Console.WriteLine("true  ^ true  = " + (true  ^ true));
16        Console.WriteLine("true  ^ false = " + (true  ^ false));
17        Console.WriteLine("false ^ true  = " + (false ^ true));
18        Console.WriteLine("false ^ false = " + (false ^ false));
19      }
20   }
```


© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\LogicalBoolTest>logicalbooltest
true  & true  = True
true  & false = False
false & true  = False
false & false = False
------------
true  | true  = True
true  | false = True
false | true  = True
false | false = False
------------
true  ^ true  = False
true  ^ false = True
false ^ true  = True
false ^ false = False

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\LogicalBoolTest>
```

Figure 6-18: Results of Running Example 6.16

Conditional AND and OR Expression Operators

The conditional operators AND '`&&`' and OR '`||`' are also referred to as the *short-circuiting* logical operators. The reason for this alternate name is that they will skip the evaluation of the second operand if the expression can be completely evaluated solely on the value of the first operand. For example, the second operand in the expression (`true || true`) can be safely skipped because the OR operator requires only one true operand. However, the second operand in the expression (`false || true`) must be evaluated since the first operand was false. These operators are demonstrated in Example 6.17. Figure 6-19 gives the results of running this program.

6.17 ConditionalOpsTest.cs

```
1    using System;
2
3    public class ConditionalOpsTest {
4       static void Main(){
5         Console.WriteLine("true  && true  = " + (true  && true));
6         Console.WriteLine("true  && false = " + (true  && false));
7         Console.WriteLine("false && true  = " + (false && true));
8         Console.WriteLine("false && false = " + (false  && false));
9         Console.WriteLine("--------------------");
10        Console.WriteLine("true  || true  = " + (true  || true));
11        Console.WriteLine("true  || false = " + (true  || false));
12        Console.WriteLine("false || true  = " + (false || true));
13        Console.WriteLine("false || false = " + (false || false));
14      }
15   }
```

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\ConditionalOperatorsTest>conditionalopstest
true  && true  = True
true  && false = False
false && true  = False
false && false = False
--------------------
true  || true  = True
true  || false = True
false || true  = True
false || false = False

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\ConditionalOperatorsTest>_
```

Figure 6-19: Results of Running Example 6.17

Conditional (Ternary) Expression Operator

The conditional operator '`?:`', also referred to as the *ternary* operator, takes one boolean operand expression. Based on the results of its evaluation, it returns one of two possible expressions. For example, consider the following ternary operator statement:

```
(boolean conditional expression) ? expression A : expression B;
```


© 2008 Rick Miller — All Rights Reserved

If the conditional expression evaluates to true, then expression A is evaluated and returned as a result of the operation. If the conditional expression evaluates to false, then expression B is evaluated and returned instead. The ternary operator never evaluates both alternate expressions. Example 6.18 shows the ternary operator in use.

6.18 TernaryOperatorTest.cs

```
1    using System;
2
3    public class TernaryOperatorTest {
4      static void Main(){
5        Console.WriteLine(true  ? "Return this string if true" : "Return this string if false");
6        Console.WriteLine(false ? "Return this string if true" : "Return this string if false");
7        Console.WriteLine();
8        int i = 3;
9        int j = 7;
10       Console.WriteLine((i < j)  ? "Return this string if true" : "Return this string if false");
11       Console.WriteLine((i > j)  ? "Return this string if true" : "Return this string if false");
12     }
13   }
```

Referring to Example 6.18 — lines 5 and 6 utilize the boolean literals "true" and "false" as arguments to the ternary operator's conditional expression. Since true is always true, the first string will be returned. On line 6, the second string will always be returned. On lines 8 and 9, two integer variables i and j are declared and initialized to the values 3 and 7, respectively. On lines 10 and 11, these variables are used to demonstrate how an actual conditional expression might be constructed.

The use of the boolean literals on lines 5 and 6 triggers a compiler warning that says it has detected unreachable code, as Figure 6-20 shows. Figure 6-21 shows the results of running the program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\TernaryOperatorTest>csc TernaryOperatorTest.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

TernaryOperatorTest.cs(5,62): warning CS0429: Unreachable expression code detected
TernaryOperatorTest.cs(6,31): warning CS0429: Unreachable expression code detected

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\TernaryOperatorTest>
```

Figure 6-20: Compiler Warning due to Unreachable Code

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\TernaryOperatorTest>ternaryoperatortest
Return this string if true
Return this string if false

Return this string if true
Return this string if false

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\TernaryOperatorTest>
```

Figure 6-21: Results of Running Example 6.18

Assignment Expression Operators

The assignment expression operators include both the simple assignment operator `'='`, which you have seen used throughout this chapter, and the compound assignment operators +=. -=. *=. /=, %=. <<=, >>=, &=, |=, and ^=. The '+=' operator is also overloaded to include event assignment, which I will cover in depth in Chapter 12.

The compound operators, as their name suggests, combine the indicated operation with an assignment. This makes for a convenient shorthand way of doing things. For example, the expression `i = i + 1` can be written `i += 1` with the help of the compound operator. Example 6.19 demonstrates the use of several compound operators. Figure 6-22 shows the results of running this program.

6.19 AssignmentOpsTest.cs

```
1    using System;
2
3    public class AssignmentOpsTest {
4      static void Main(){
5        int i = 0;
6        Console.WriteLine("The value of i initially = " + i);
7        Console.WriteLine("i += 1 = " + (i += 1));
8        Console.WriteLine("i -= 1 = " + (i -= 1));
```

 © 2008 Rick Miller — All Rights Reserved

```
9         Console.WriteLine("i += 2 = " + (i += 2));
10        Console.WriteLine("i *= 2 = " + (i *= 2));
11        Console.WriteLine("i /= 2 = " + (i /= 2));
12      }
13    }
```

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\AssignmentOpsTest>assignmentopstest
The value of i initially = 0
i += 1 = 1
i -= 1 = 0
i += 2 = 2
i *= 2 = 4
i /= 2 = 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter6\AssignmentOpsTest>
```

Figure 6-22: Results of Running Example 6.19

Quick Review

Statements are the fundamental building blocks of C# programs. A statement can be thought of as the smallest standalone element of a program. Programs are built using sequences of statements. C# offers nineteen different types of statements.

The term *operator precedence* refers to the order in which the C# compiler evaluates the operators appearing in an expression statement. The term *associativity* refers to the direction in which the C# compiler performs a series of operations.

Summary

A simple C# application is a class that contains a Main() method. The purpose of the Main() method is to provide an entry point for program execution. There are four authorized versions of the Main() method; each version has a different method signature.

A class that contains a Main() method can be compiled into an executable assembly. A class with no Main() method can be compiled into a module. Modules can be added to assemblies. Modules created in CLI-compliant languages other than C# can be compiled with C# modules to form executable assemblies.

Identifiers are sequences of characters that represent names of objects in a program. Identifiers are used to name classes, structures, methods, variables, constants, properties, fields, enums, etc. Identifiers can start with either an uppercase or lowercase letter or an underscore '_' character followed by any number of letters, digits, and underscores.

Reserved keywords are identifiers that have special meaning within the C# language. You cannot reintroduce a reserved keyword as a name for an object within your program. You can, however, prepend the '@' symbol to a reserved keyword to formulate a valid identifier, however, I discourage doing this as it renders code hard to read, understand, and maintain.

C# has two kinds of types: value types and reference types. Value type variables contain the actual data as defined by the type. Reference type variables contain a reference to an object in memory. Two or more reference type variables could reference the same object in memory. The C# predefined types map to structures within the System namespace. System.object is the base type for all other types.

Skill-Building Exercises

1. **API Drill:** Visit the Microsoft Developer Network (MSDN) [www.msdn.com] and research the C# predefined type structures located in the System namespace. Use Table 6-2 as a guide. Take note of the methods and fields each

© 2008 Rick Miller — All Rights Reserved

structure makes available for use. Track down and study any interfaces these structures may implement.

2. **Practice Makes Perfect:** Compile and run each of the example programs listed in this chapter.

3. **Type Ranges:** Write a program that displays to the console a list of the predefined numeric types and shows their minimum and maximum values. **Hint:** Pay attention to what you discovered in Skill-Building Exercise #1!

Suggested Projects

1. **Average Five Numbers:** Write a program that computes the average of five floating point numbers and writes the answer to the console.

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 compares the values of two integer variables 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 computes the *time* required to travel a given *distance* in miles at a certain *speed* in miles/hour. The equation required is:

$$t = d/s$$

5. **Compute Average Speed:** Write a program that computes the *speed* required to travel a certain *distance* in a given amount of *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. Write the results to the console.

7. **Division By Shifting:** Write a program that divides an integer by 2 using the right shift operator. Explain why shifting a number to the right performs a division. What happens when you shift a number to the left? **Hint:** Think in terms of binary digits.

Self-Test Questions

1. What two kinds of types does C# support?

2. How many predefined types does C# support.

3. Describe in your own words how two reference type variables might end up referencing the same object.

4. What’s the difference between a value type and a reference type?

5. What character is used to terminate a statement?

6. What’s the purpose of the *new* operator.

 © 2008 Rick Miller — All Rights Reserved

7. How many different forms of the Main() method does C# support?

8. What's the purpose of a Main() method?

9. What's the purpose of the *using* directive.

10. Can a reserved keyword be used as an identifier? Explain your answer.

References

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Notes

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Chapter 7

Contax T / Kodak Tri-X

Amsterdam

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" statements
- Describe the purpose and use of the "goto" statement
- Describe the purpose and use of the "try/catch" statement
- Demonstrate your ability to use control-flow statements in simple C# programs

 © 2008 Rick Miller — All Rights Reserved

Introduction

Program control-flow statements are an important part of the C# 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 evaluation of a conditional expression. 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`. There is also a `foreach` statement that is used with arrays and collections. I will therefore postpone coverage of the `foreach` statement until Chapter 8.

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. This will enable you to write increasingly powerful programs.

In addition to selection and iteration statements, I will show you how to use the keywords `break`, `continue`, and `goto`. 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 C# 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 C# 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 conditionally executes a block of code based on the evaluation of a conditional expression. Figure 7-1 graphically shows what happens during the processing of an `if` statement.

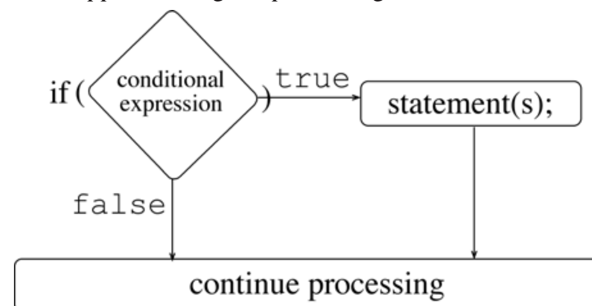


Figure 7-1: if Statement Execution Diagram

A *conditional expression* evaluates to either *true* or *false*. 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 program bypasses the statements contained within the `if` statement body and processing continues with 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.

7.1 IfStatementTest.cs

```
1    using System;
2
3    public class IfStatementTest {
4      static void Main(String[] args){
```

 © 2008 Rick Miller — All Rights Reserved

```
5        int int_i = Convert.ToInt32(args[0]);
6        int int_j = Convert.ToInt32(args[1]);
7
8        if(int_i < int_j)
9          Console.WriteLine(int_i + " is less than " + int_j);
10     }
11   }
```

Referring to Example 7.1 — the program converts command-line input from strings to integers with the help of the System.Convert.ToInt32() method and assigns the resultant values to the variables int_i and int_j on lines 5 and 6. (Study the Convert class for more conversion methods.) The `if` statement begins on line 8. The less-than operator ‘<’ compares 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 9 executes. If not, line 9 is skipped and the program exits. Figure 7-2 shows the results of running this program.

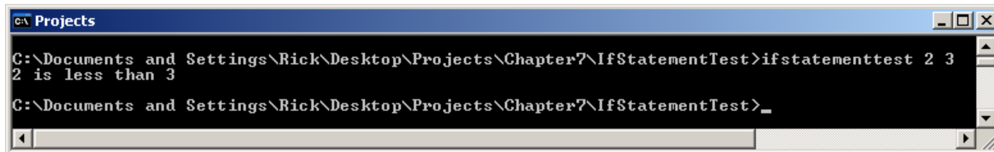


Figure 7-2: Results of Running Example 7.1

Handling Program Error Conditions

If you were unlucky enough to forget to supply two properly-formed integer values on the command-line when you ran the previous program you would have received an error message like the one shown in Figure 7-3.

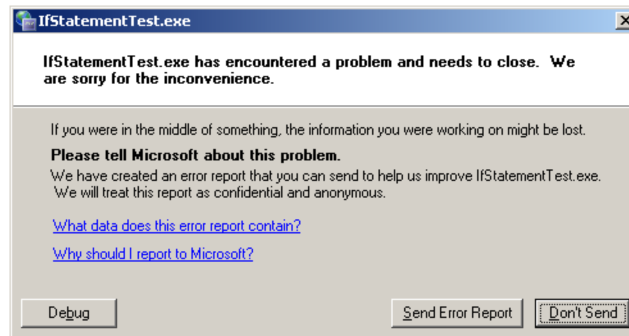


Figure 7-3: Typical .NET Error Message Dialog Window

This dialog window signals to you that an error condition or exception of some kind has occurred in your program, and it has stopped running. (It has stopped running rather abruptly I might add!) Since you're running a program you wrote, click the **Don't Send** button.

As it turns out, there are several things that can go wrong with this program. Failure to supply the proper number of arguments on the command line will cause an IndexOutOfRangeException. This type of exception occurs if you attempt to access an array element that does not exist. Figure 7-4 shows the exception message output to the console for this type of exception.

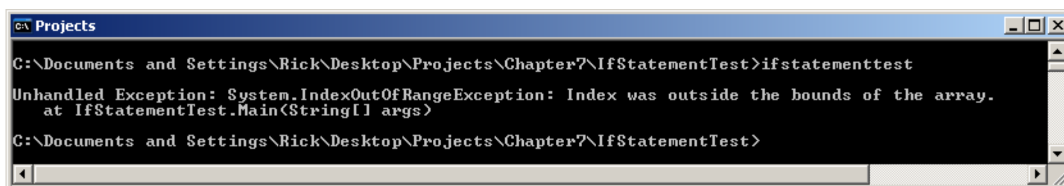


Figure 7-4: Unhandled IndexOutOfRangeException Message

You will receive a different error message if you supply the right number of arguments, but one of the arguments fails to convert properly to an integer value. Figure 7-5 shows the results of running Example 7.1 with one good and one bad input argument.

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementTest>ifstatementtest 1 g

Unhandled Exception: System.FormatException: Input string was not in a correct format.
   at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number, NumberFormatInfo info, Boolea
n parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   at IfStatementTest.Main(String[] args)

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementTest>_
```

Figure 7-5: FormatException Error Message

This time the program threw a FormatException. To produce this output, the arguments '1' and 'g' were used as input to the program. The character 'g' failed to convert to an integer and caused the exception condition.

Catching Exceptions With Try/Catch Blocks

Any code that could potentially throw an exception when executed should appear within a `try/catch` statement, also referred to as a *try/catch block*. Doing this places you in control of your program. Your code may still throw an exception, but if it's in a `try/catch` block, you can properly handle the exception and gracefully recover. The user may never know the exception occurred. Look at the code in Example 7.2.

7.2 IfStatementTest.cs (Mod 1)

```
1    using System;
2
3    public class IfStatementTest {
4      static void Main(String[] args){
5
6        try{
7          int int_i = Convert.ToInt32(args[0]);
8          int int_j = Convert.ToInt32(args[1]);
9
10         if(int_i < int_j)
11           Console.WriteLine(int_i + " is less than " + int_j);
12
13       }catch(IndexOutOfRangeException){
14         Console.WriteLine("You must enter two integer numbers! Please try again.");
15       }catch(FormatException){
16         Console.Write("One of the arguments you entered was not a valid integer value! ");
17         Console.WriteLine("Please try again.");
18       }
19     }
20   }
```

Referring to Example 7.2 — the body of the original program has been placed inside the body of a `try` block. You can think of the `try` block as a *guarded region*. If everything placed within the `try` block executes, great! If not, the `try` block will exit at the point where the exception is thrown, and the rest of the code within the `try` block will be skipped.

In Example 7.2, all of the code from Example 7.1 inside the body of the Main() method, including the `if` statement, was placed inside the body of the `try` block. Doing this avoids executing the `if` statement if either error condition is encountered since the values of int_i and int_j will not have been properly initialized.

A `try` block can be followed by one or more `catch` blocks. If the program throws an exception, the `catch` block assumes execution based on the type of exception thrown. This is referred to as *handling* the exception. The first `catch` block begins on line 13. It's catching the IndexOutOfRangeException. If this exception occurs, then a message is printed to the screen reminding the user to enter two valid numbers. The second `catch` block handles the FormatException, which will occur if either of the command-line arguments cannot be converted to integers.

Figure 7-6 shows the results of running Example 7.2 with various types of bad input.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod1>ifstatementtest
You must enter two integer numbers! Please try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod1>ifstatementtest 1 g
One of the arguments you entered was not a valid integer value! Please try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod1>ifstatementtest 1 2
Yes 1 is less than 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod1>_
```

Figure 7-6: Results of Running Example 7.2

© 2008 Rick Miller — All Rights Reserved

Referring to Figure 7-6 — notice that now when you forget to supply arguments to the program, you get the gentle reminder. Also, if one or more of the supplied arguments fail to convert to an integer value, you get an appropriate message.

Discovering What Methods Throw What Exceptions

How do you know what type of an exception a .NET Framework API method might throw? You guessed it. You'll have to look it up in the API documentation.

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 7.3 gives an example of such a code block.

7.3 IfStatementTest.cs (Mod 2)

```
1    using System;
2
3    public class IfStatementTest {
4      static void Main(String[] args){
5
6        try{
7          int int_i = Convert.ToInt32(args[0]);
8          int int_j = Convert.ToInt32(args[1]);
9
10         if(int_i < int_j) {
11           Console.Write("Yes ");
12           Console.WriteLine(int_i + " is less than " + int_j);
13         }
14
15       }catch(IndexOutOfRangeException){
16         Console.WriteLine("You must enter two integer numbers! Please try again.");
17       }catch(FormatException){
18         Console.Write("One of the arguments you entered was not a valid integer value! ");
19         Console.WriteLine("Please try again.");
20       }
21     }
22   }
```

Referring to Example 7.3 — notice that now 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 10 and ends with the closing brace on line 13. Figure 7-7 shows the results of running this program.

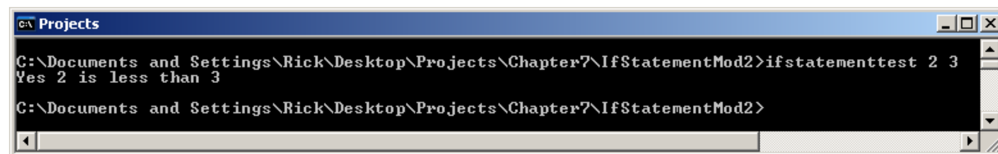


Figure 7-7: Results of Running Example 7.3

Executing Consecutive If Statements

You can follow one `if` statement with another as is shown in Example 7.4.

7.4 IfStatementTest.cs (Mod 3)

```
1    using System;
2
3    public class IfStatementTest {
4      static void Main(String[] args){
5
6        try{
7       int int_i = Convert.ToInt32(args[0]);
8       int int_j = Convert.ToInt32(args[1]);
9
10      if(int_i < int_j) {
11         Console.Write("Yes ");
12         Console.WriteLine(int_i + " is less than " + int_j);
13         }
14
```


© 2008 Rick Miller — All Rights Reserved

```
15       if(int_i > int_j) {
16          Console.Write("No ");
17          Console.WriteLine(int_i + " is greater than " + int_j);
18       }
19
20
21      }catch(IndexOutOfRangeException){
22        Console.WriteLine("You must enter two integer numbers! Please try again.");
23      }catch(FormatException){
24        Console.Write("One of the arguments you entered was not a valid integer value!");
25        Console.WriteLine("Please try again.");
26      }
27     }
28   }
```

Referrring to Example 7.4 — when this program executes, the program evaluates the expressions of both `if` statements. When the program is run with the inputs 2 and 3, the expression of the first `if` statement on line 10 evaluates to true and its body statements execute. The second `if` statement's expression evaluates to false and its body statements are skipped.

The opposite happens when the program is run a second time using input values 3 and 2. 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 execute. Figure 7-8 shows the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod3>ifstatementtest 2 3
Yes 2 is less than 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod3>ifstatementtest 3 2
No 3 is greater than 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod3>_
```

Figure 7-8: Results of Running Example 7.4

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 7-9 provides an execution diagram of the `if/else` statement.

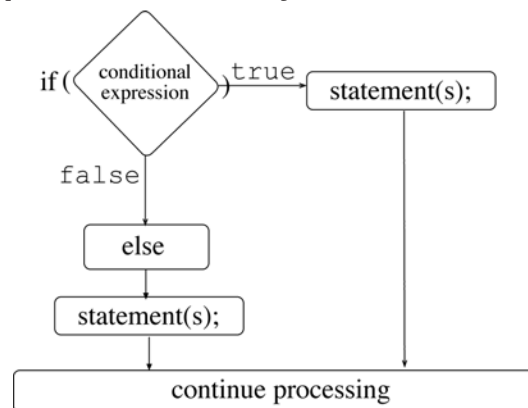


Figure 7-9: if/else Statement Execution Diagram

The `if/else` statement behaves like the `if` statement, except that now when the expression evaluates to false, the statements contained within the body of the `else` clause execute. Example 7.5 provides the same functionality as Example 7.4 using one `if/else` statement.

7.5 IfElseStatementTest.cs

```
1    using System;
2
3    public class IfElseStatementTest {
4      public static void Main(String[] args){
5
6      try{
7       int int_i = Convert.ToInt32(args[0]);
8       int int_j = Convert.ToInt32(args[1]);
9
```

 © 2008 Rick Miller — All Rights Reserved

```
10       if(int_i < int_j) {
11         Console.Write("Yes ");
12         Console.WriteLine(int_i + " is less than " + int_j);
13         } else {
14            Console.Write("No ");
15            Console.WriteLine(int_i + " is not less than " + int_j);
16            }
17        }catch(IndexOutOfRangeException){
18         Console.WriteLine("You must enter two integer numbers! Please try again.");
19       }catch(FormatException){
20         Console.Write("One of the arguments you entered was not a valid integer value!");
21         Console.WriteLine("Please try again.");
22       }
23     }
24   }
```

Referring to Example 7.5 — the `if` statement begins on line 10. Should the expression evaluate to true, the code block that forms the body of the `if` statement executes. Should the expression evaluate to false, the code block following the `else` keyword executes. Figure 7-10 shows the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfElseStatementTest>ifelsestatementtest 1 2
Yes 1 is less than 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfElseStatementTest>ifelsestatementtest 2 1
No 2 is not less than 1

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfElseStatementTest>_
```

Figure 7-10: Results of Running Example 7.5

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 7.6 illustrates the use of chained `if/else` statements in a program.

7.6 ChainedIfElseTest.cs

```
1    using System;
2
3    public class ChainedIfElseTest {
4      public static void Main(String[] args){
5
6        try{
7          int int_i = Convert.ToInt32(args[0]);
8          int int_j = Convert.ToInt32(args[1]);
9
10         if(int_i < int_j) {
11           Console.Write("Yes ");
12           Console.WriteLine(int_i + " is less than " + int_j);
13          } else if(int_i == int_j) {
14                  Console.Write("Exact match! ");
15                  Console.WriteLine(int_i + " is equal to " + int_j);
16                else{
17                   Console.Write("No ");
18                   Console.WriteLine(int_i + " is greater than " + int_j);
19                }
20        }catch(IndexOutOfRangeException){
21             Console.WriteLine("You must enter two integer numbers! Please try again.");
22        }catch(FormatException){
23             Console.Write("One of the arguments you entered was not a valid integer value!");
24             Console.WriteLine("Please try again.");
25        }
26      }
27   }
```

There are a couple of important points to note regarding Example 7.6. First, notice how the second `if/else` statement begins on line 13 immediately following the `else` keyword of the first `if/else` statement. Second, notice how indenting is used to aid readability. Figure 7-11 gives the results of running this program.

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ChainedIfElseStatement>chainedifelsetest 1 2
Yes 1 is less than 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ChainedIfElseStatement>chainedifelsetest 1 1
Exact match! 1 is equal to 1

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ChainedIfElseStatement>chainedifelsetest 2 1
No 2 is greater than 1

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ChainedIfElseStatement>
```

Figure 7-11: Results of Running Example 7.6

Switch Statement

Use the `switch` statement, also referred to as the `switch/case` statement, in situations where you need to provide multiple execution paths based on the evaluation of a particular *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, *string*, or *enum* type value. Figure 7-12 gives the execution diagram for a `switch` statement.

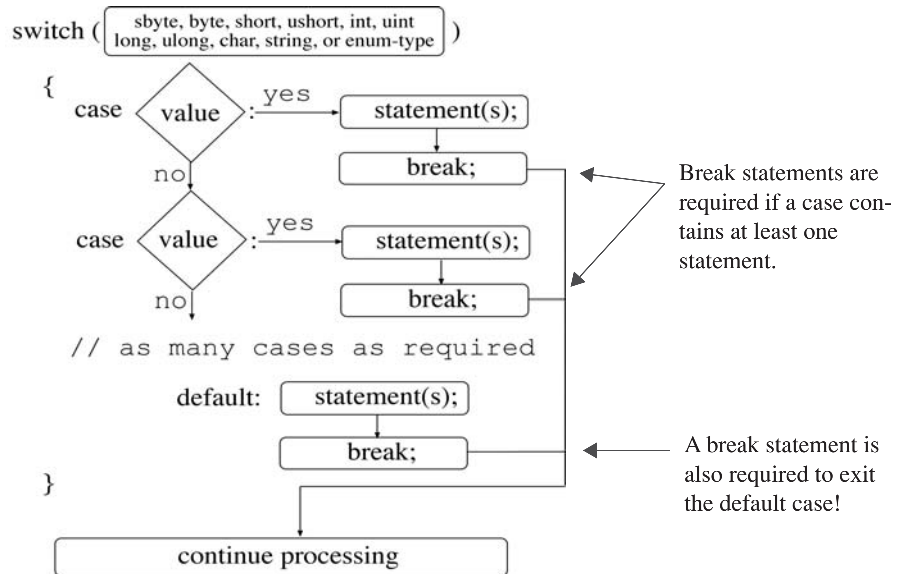


Figure 7-12: `switch` Statement Execution Diagram

When you write a `switch` statement, you will add one or more `case` clauses to it. When the `switch` statement executes, the program compares the value supplied in the parentheses of the `switch` statement to each `case` value. A match results in the execution of that `case`'s code block.

Notice in Figure 7-12 how each `case`'s statements are followed by a `break` statement. The `break` statement exits the `switch` and continues processing. The `break` statement is required because `case` statements are not allowed to implicitly continue processing (*i.e.*, fall through) to the next `case` after executing one or more statements. Example 7.7 gives an example of the `switch` statement in action.

7.7 SwitchStatementTest.cs

```
1    using System;
2
3    public class SwitchStatementTest {
4      public static void Main(String[] args){
5        try{
6           int int_i = Convert.ToInt32(args[0]);
7
8           switch(int_i){
9             case 1  : Console.WriteLine("You entered one");
10                      break;
11            case 2  : Console.WriteLine("You entered two");
12                      break;
13            case 3  : Console.WriteLine("You entered three");
14                      break;
15            case 4  : Console.WriteLine("You entered four");
16                      break;
```


© 2008 Rick Miller — All Rights Reserved

```
17          case 5  : Console.WriteLine("You entered five");
18                    break;
19          default : Console.WriteLine("Please enter a number between 1 and 5");
20                    break;
21          }
22      }catch(IndexOutOfRangeException){
23         Console.WriteLine("Enter one number at the command-line!");
24      }catch(FormatException){
25         Console.WriteLine("You entered an invalid number! Try again.");
26      }
27    }
28  }
```

Referring to Example 7.7 — this program reads a string from the command line and converts it to an integer value using our friend the Convert.ToInt32() 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 prints to the console. If there's no match, then the `default case` executes and prompts the user to enter a valid value. The `try/catch` statement handles anticipated exceptions. Figure 7-13 shows the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTest>switchstatementtest
Enter one number at the command-line!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTest>switchstatementtest h
You entered an invalid number! Try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTest>switchstatementtest 6
Please enter a number between 1 and 5

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTest>switchstatementtest 2
You entered two

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTest>_
```

Figure 7-13: Results of Running Example 7.7

Implicit Case Fall-Through

You can rewrite the `switch` statement shown in Example 7.7 to take advantage of implicit `case` fall-through. In this version, the `switch` statement relies on the help of an array, as you will see from studying the code shown in Example 7.8.

7.8 SwitchStatementTest.cs (Mod 1)

```
1    using System;
2
3    public class SwitchStatementTest {
4      public static void Main(String[] args){
5        try{
6          int int_i = Convert.ToInt32(args[0]);
7          string[] string_array = {"one", "two", "three", "four", "five"};
8
9          switch(int_i){
10           case 1  : // this works because these cases contain no statements...
11           case 2  :
12           case 3  :
13           case 4  :
14           case 5  : Console.WriteLine("You entered " + string_array[int_i-1]);
15                     break;
16           default : Console.WriteLine("Please enter a number between 1 and 5");
17                     break;
18           }
19         }catch(IndexOutOfRangeException){
20            Console.WriteLine("Enter one number at the command-line!");
21         }catch(FormatException){
22            Console.WriteLine("You entered an invalid number! Try again.");
23         }
24     }
25   }
```

Referring to Example 7.8 — although arrays are formally covered in Chapter 8, the use of one in this particular example should not be too confusing to you. A string array, similar to that used as an argument to the Main() method is declared on line 7. It is initialized to hold five string values: "one", "two", "three", "four", and "five". Each element of the array is accessed with an integer index value of between 0 and 4 where 0 represents the first element of the

© 2008 Rick Miller — All Rights Reserved

array and 4 represents the last element. The `switch` statement is then rewritten in a more streamlined fashion with the help of implicit `case` fall-through. Implicit `case` fall-through works as long as a `case` contains no statements.

The string_array variable on line 14 uses the variable int_i to provide the text representation of the numbers 1, 2, 3, 4, and 5. The variable int_i serves as the array index. Notice, however, that 1 must be subtracted from int_i to yield the proper array offset and refer to the correct string_array element.

If you are confused by the use of the array in this example don't panic. I cover arrays in detail in Chapter 8. Figure 7-14 gives the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTestMod1>switchstatementtest
Enter one number at the command-line!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTestMod1>switchstatementtest j
You entered an invalid number! Try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTestMod1>switchstatementtest 7
Please enter a number between 1 and 5

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTestMod1>switchstatementtest 3
You entered three

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTestMod1>
```

Figure 7-14: Results of Running Example 7.8

Nested Switch Statement

Switch statements can be nested to yield complex programming logic. Study Example 7.9.

7.9 NestedSwitchTest.cs

```
1    using System;
2
3    public class NestedSwitchTest {
4      public static void Main(String[] args){
5        try{
6          char char_c = (args[0])[0];
7          int int_i = Convert.ToInt32(args[1]);
8
9          switch(char_c){
10           case 'U' :
11           case 'u' : switch(int_i){
12                        case 1:
13                        case 2:
14                        case 3:
15                        case 4:
16                        case 5: Console.WriteLine("You entered " + char_c + " and " + int_i);
17                                break;
18                        default: Console.WriteLine("Please enter: 1, 2, 3, 4, or 5");
19                                 break;
20                      }
21                      break;
22           case 'D' :
23           case 'd' : switch(int_i){
24                        case 1:
25                        case 2:
26                        case 3:
27                        case 4:
28                        case 5: Console.WriteLine("You entered " + char_c + " and " + int_i);
29                                break;
30                        default: Console.WriteLine("Please enter: 1, 2, 3, 4, or 5");
31                                 break;
32                      }
33                      break;
34             default: Console.WriteLine("Please enter: U, u, D, or d");
35                      break;
36           }
37
38         }catch(IndexOutOfRangeException){
39             Console.WriteLine("The program requires two arguments! Please try again.");
40         }catch(FormatException){
41             Console.WriteLine("Invalid number. Please try again.");
42         }
43       }
44     }
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 7.9 — 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 integer 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 readers of the code distinguish between outer and inner `switch` statements. Figure 7-15 gives the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest
The program requires two arguments! Please Try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest u 2
You entered u and 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest U 3
You entered U and 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest 4 2
Please enter: U, u, D, or d

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest D u
Invalid number. Please try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest D 3
You entered D and 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>
```

Figure 7-15: Results of Running Example 7.9

Quick Review

Selection statements provide alternative program execution paths based on the evaluation of a conditional expression. 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 an *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, *string*, or *enum* 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. Implicit case fall-through is only allowed if a case contains no statements.

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.` I discuss these statements in detail in this section. Iteration statements are also referred to as loops. So, when you hear other programmers talking about a `for` loop, `while` loop, or `do` loop, they are referring to the iteration statements.

While Statement

The `while` statement repeats one or more program statements based on the results of an expression evaluation. Figure 7-16 shows the execution diagram for the `while` statement.

Personality Of The While Statement

The `while` statement has the following personality:

- It evaluates 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 you want a `while` loop to repeat forever until some action inside it forces it to exit.)

© 2008 Rick Miller — All Rights Reserved

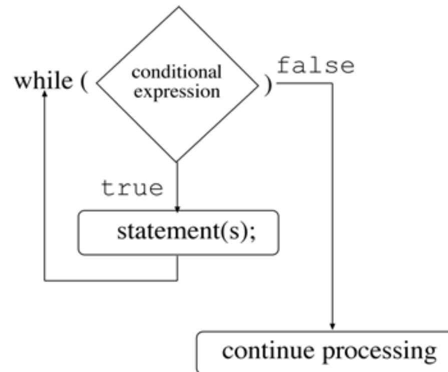


Figure 7-16: while Statement Execution Diagram

• The expression may evaluate to false on the first try and therefore not execute its body statements.

Essentially, the `while` loop performs the expression evaluation first, then executes its body statements. To prevent a `while` loop from looping indefinitely, a program statement somewhere within the body of the `while` loop 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 7.10 shows the `while` statement in action.

7.10 WhileStatementTest.cs

```
1    using System;
2
3    public class WhileStatementTest {
4       static void Main(){
5          int int_i = 0;
6
7          while(int_i < 10){
8            Console.WriteLine("The value of int_i = " + int_i);
9            int_i++;
10         }
11      }
12   }
```

Referring to Example 7.10 — on line 5, the integer variable int_i is declared and initialized to 0. The variable int_i is then 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 execute. (This includes all statements between the opening brace appearing at the end of line 7 and the closing brace on line 10.) Notice how the value of int_i is incremented with the ‘`++`’ operator. This is an essential step. If int_i is not incremented, the expression will always evaluate to true, which would result in an infinite loop. Figure 7-17 shows the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\WhileStatementTest>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

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\WhileStatementTest>_
```

Figure 7-17: Results of Running Example 7.10

Do/While Statement

The `do/while` statement repeats one or more body statements based on the result of a conditional expression evaluation. The `do/while` loop differs from the `while` loop in that its body statements execute at least once before the expression is evaluated. Figure 7-18 gives the execution diagram for a `do/while` statement.

 © 2008 Rick Miller — All Rights Reserved

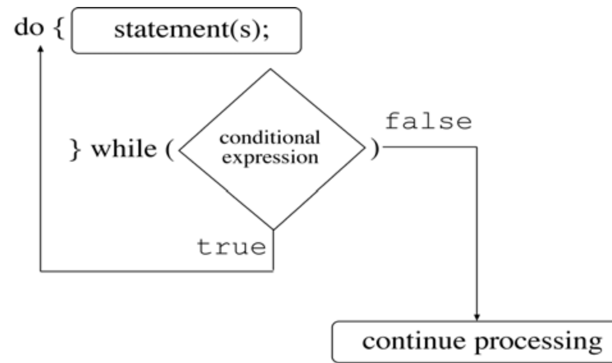


Figure 7-18: do/while Statement Execution Diagram

Personality Of The Do/While Statement

The `do/while` statement has the following personality:

- It executes its body statements once before evaluating its expression.
- A statement within its body must either take some action that will make the expression evaluate to false or explicitly exit the loop, otherwise the `do/while` loop will repeat forever. (And, like the while loop, sometimes you may want it to repeat forever.)
- Use a `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 when the expression evaluation occurs. The `while` statement evaluates it at the beginning — the `do/while` statement evaluates it at the end. Example 7.11 shows the `do/while` statement in action.

7.11 DoWhileStatementTest.cs

```
1    using System;
2
3    public class DoWhileStatementTest {
4      static void Main(){
5        int int_i = 0;
6
7         do {
8           Console.WriteLine("The value of int_i = " + int_i);
9           int_i++;
10        }while(int_i < 10);
11     }
12   }
```

Referring to Example 7.11 — on line 5, the integer variable int_i is declared and initialized to 0. The `do/while` statement begins on line 7, and its body statements are contained between the opening brace appearing at the end of line 7 and the closing brace on line 10. Notice that the `while` keyword and its expression are terminated with a semicolon. Figure 7-19 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\DoWhileStatementTest>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

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\DoWhileStatementTest>_
```

Figure 7-19: Results of Running Example 7-11

 © 2008 Rick Miller — All Rights Reserved

For Statement

The `for` statement repeats 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 7.20 gives the execution diagram for the `for` statement.

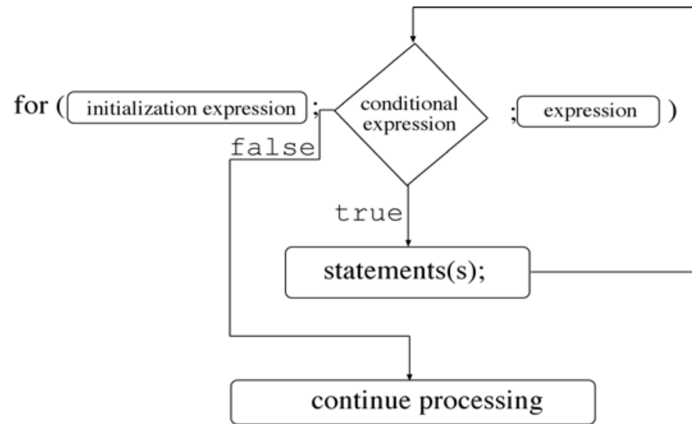


Figure 7-20: for Statement Execution Diagram

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 its body statements execute.

Personality Of The For Statement

The `for` statement has the following personality:

- It provides a convenient way to initialize counter variables, perform conditional expression evaluation, and increment loop-control variables.
- The conditional expression is evaluated up front before its code block executes, 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 7.12 shows the `for` statement in action.

7.12 ForStatementTest.cs

```
1    using System;
2
3    public class ForStatementTest {
4      static void Main(){
5
6          for(int i = 0; i < 10; i++){
7            Console.WriteLine("The value of i = " + i);
8          }
9       }
10   }
```

Referring to Example 7.12 — the `for` statement begins on line 6. 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 only executes one statement, you can omit the braces. (This is true

© 2008 Rick Miller — All Rights Reserved

for the `while` and `do` loops as well.) For example, the `for` statement shown in Example 7.12 could be rewritten in the following manner:

```
for(int i = 0; i < 10; i++)
  Console.WriteLine("The value of i = " + i );
```

Figure 7-21 shows the results of running example 7.12.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ForStatementTest>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

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ForStatementTest>
```

Figure 7-21: Results of Running Example 7.12

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:

$$\sum_{i=1}^{m}\left(\sum_{j=1}^{n}(i \times j)\right)$$

Example 7.13 offers one possible solution for this particular problem.

7.13 NestedForLoop.cs

```
1    using System;
2
3    public class NestedForLoop {
4      static void Main(String[] args){
5        try{
6          int limit_i = Convert.ToInt32(args[0]);
7          int limit_j = Convert.ToInt32(args[1]);
8          int total = 0;
9
10         for(int i = 1; i <= limit_i; i++){
11           for(int j = 1; j <= limit_j; j++){
12             total += (i*j);
13           }
14         }
15         Console.WriteLine("The total is: " + total);
16       }catch(IndexOutOfRangeException){
17          Console.WriteLine("Two arguments are required to run this program!");
18       }catch(FormatException){
19          Console.WriteLine("Both arguments must be integers!");
20       }
21     }
22   }
23
```

Referring to Example 7.13 — 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 compared against the values of i and j, respectively, in the middle expression of each `for` loop. Notice, too, that indenting makes it easier to distinguish between the outer and inner `for` statements. Figure 7-22 shows the results of running this program using various inputs.

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedForLoop>nestedforloop 1 2
The total is: 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedForLoop>nestedforloop 2 2
The total is: 9

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedForLoop>nestedforloop 4 4
The total is: 100

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedForLoop>nestedforloop 5 5
The total is: 225

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedForLoop>
```

Figure 7-22: Results of Running Example 7.13

Mixing Selection And 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 7.14 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 Convert class. It also uses `try/catch` blocks like the previous examples.

7.14 CheckBookBalancer.cs

```
1     using System;
2
3     public class CheckBookBalancer {
4         static void Main(){
5           /**** Initialize Program Variables ******/
6
7           char keep_going = 'Y';
8           double balance = 0.0;
9           double deposits = 0.0;
10          double withdrawals = 0.0;
11          bool good_double = false;
12
13          /**** Display Welcome Message ****/
14          Console.WriteLine("Welcome to Checkbook Balancer");
15
16
17          /**** Get Starting Balance *****/
18          do{
19           try{
20             Console.Write("Please enter the opening balance: ");
21             balance = Convert.ToDouble(Console.ReadLine());
22             good_double = true;
23             }catch(FormatException){ Console.WriteLine("Please enter a valid balance!");}
24            }while(!good_double);
25
26
27          /**** Add All Deposits ****/
28          while((keep_going == 'y') || (keep_going == 'Y')){
29              good_double = false;
30              do{
31                try{
32                   Console.Write("Enter a deposit amount: ");
33                   deposits += Convert.ToDouble(Console.ReadLine());
34                   good_double = true;
35                   }catch(FormatException){ Console.WriteLine("Please enter a valid deposit value!");}
36                 }while(!good_double);
37               Console.WriteLine("Do you have to enter another deposit? y/n");
38               try{
39                 keep_going = (Console.ReadLine())[0];
40                  }catch(IndexOutOfRangeException){ Console.WriteLine("Problem reading input!");}
41          }
42
43         /**** Subtract All Checks Written ****/
44          keep_going = 'y';
45          while((keep_going == 'y') || (keep_going == 'Y')){
46              good_double = false;
47              do{
48                try{
49                   Console.Write("Enter a check amount: ");
50                   withdrawals += Convert.ToDouble(Console.ReadLine());
```

 © 2008 Rick Miller — All Rights Reserved

```
51                  good_double = true;
52                  }catch(FormatException){ Console.WriteLine("Please enter a valid check amount!");}
53                }while(!good_double);
54              Console.WriteLine("Do you have to enter another check? y/n");
55              try{
56                keep_going = (Console.ReadLine())[0];
57                 }catch(IndexOutOfRangeException){ Console.WriteLine("Problem reading input!");}
58          }
59
60          /**** Display Final Tally ****/
61
62          Console.WriteLine("***************************************");
63          Console.WriteLine("Opening balance:     $    " + balance);
64          Console.WriteLine("Total deposits:     +$    " + deposits);
65          Console.WriteLine("Total withdrawals:  -$    " + withdrawals);
66          balance = balance + (deposits - withdrawals);
67          Console.WriteLine("New balance is:      $    " + balance);
68          Console.WriteLine("\n\n");
69      }
70    }
```

Figure 7-23 shows the results of running the CheckBookBalancer program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\CombinationStatements>checkbookbalancer
Welcome to Checkbook Balancer
Please enter the opening balance: 500.00
Enter a deposit amount: 23.45
Do you have to enter another deposit? y/n
y
Enter a deposit amount: 45.00
Do you have to enter another deposit? y/n
n
Enter a check amount: 12.23
Do you have to enter another check? y/n
y
Enter a check amount: 10.20
Do you have to enter another check? y/n
n
***************************************
Opening balance:     $    500
Total deposits:     +$    68.45
Total withdrawals:  -$    22.43
New balance is:      $    546.02


C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\CombinationStatements>_
```

Figure 7-23: 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-counter variable declaration and initialization, conditional expression evaluation, and loop-counter variable incrementing in one statement.

Break, Continue, And Goto

The `break`, `continue`, and `goto` statements belong to a class of C# statements known as *jump* statements. 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. The `continue` statement stops the processing of the current loop iteration and begins the next iteration. It is used in conjunction with `for`, `while`, and `do` loops. The `goto` statement is used with a label to jump to a specific point in a program.

© 2008 Rick Miller — All Rights Reserved

Break Statement

An `break` statement looks like this:

```
break;
```

Use a `break` statement in the body of `switch`, `for`, `while`, and `do/while` statements to immediately exit its containing statement. When used in a nested statement structure, the `break` exits the innermost containing statement. To exit nested statement structures, you must use the `goto` statement instead. Example 7.15 shows the `break` statement in action.

7.15 BreakStatementTest.cs

```
1    using System;
2
3    public class BreakStatementTest {
4      static void Main(){
5         for(int i = 0; i < 2; i++){
6             for(int j = 0; j < 1000; j++){
7               Console.WriteLine("Inner for loop - j = " + j);
8               if(j == 3) break;
9             }
10          Console.WriteLine("Outer for loop - i = " + i);
11        }
12     }
13   }
```

Referring to example 7.15 — here, a `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 8 checks the value of j. If j == 3, the `break` statement executes, otherwise the loop is allowed to continue. As soon as the expression j == 3 evaluates to *true*, the inner `for` loop terminates and the outer `for` loop executes the Console.WriteLine() method on line 10, and then begins another iteration. Figure 7-24 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\BreakStatementTest>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

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\BreakStatementTest>_
```

Figure 7-24: Results of Running Example 7.15

Continue Statement

The `continue` statement stops the current iteration of its containing loop and begins a new iteration. Example 7.16 shows the `continue` statement in action in a short program that prints odd integers.

7.16 ContinueStatementTest.cs

```
1    using System;
2
3    public class ContinueStatementTest {
4      static void Main(String[] args){
5        try{
6          int limit_i = Convert.ToInt32(args[0]);
7          for(int i = 0; i<limit_i; i++){
8            if((i % 2) == 0) continue;
9            Console.WriteLine(i);
10         }
11       }catch(IndexOutOfRangeException){
12          Console.WriteLine("This program requires one integer argument!");
13       }catch(FormatException){
14          Console.WriteLine("Argument must be a valid integer!");
15       }
16     }
17   }
```

 © 2008 Rick Miller — All Rights Reserved

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 8 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 executes 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. Figure 7-25 shows the results of running this program with the input 24.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ContinueStatementTest>continuestatementtest 24
1
3
5
7
9
11
13
15
17
19
21
23

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ContinueStatementTest>
```

Figure 7-25: Results of Running Example 7.16

Goto Statement

The `goto` statement is used with a label to jump to the labeled point in the program. Example 7.17 shows the `goto` statement being used to implement a simple repetitive loop.

7.17 GotoStatementTest.cs

```
1    using System;
2
3    public class GotoStatementTest {
4      static void Main(){
5        int i = 0;
6        Label1: Console.WriteLine("Label1 statement " + i++);
7        if(i < 10) goto Label1;
8      }
9    }
```

Referring to Example 7.17 — this program shows how the `goto` statement can be used to implement a `do/while` statement. The statement on line 6 prints first and then increments the variable i. The `if` statement on line 7 compares i to the value 10. If i < 10, then execution jumps to Label1 on line 6. When i reaches 10, the program exits. Figure 7-26 shows the results of running this program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\GotoStatement>gotostatementtest
Label1 statement 0
Label1 statement 1
Label1 statement 2
Label1 statement 3
Label1 statement 4
Label1 statement 5
Label1 statement 6
Label1 statement 7
Label1 statement 8
Label1 statement 9

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\GotoStatement>
```

Figure 7-26: Results of Running Example 7.17

Quick Review

The `break` and `continue` statements provide fine-grained control over iteration statements. In addition to exiting `switch` statements, the `break` statement is used to exit `for`, `while`, and `do/while` loops. The `continue` statement terminates the current iteration of the loop it's embedded within and forces the start of a new iteration. The `goto` statement is used with a label to jump to the labeled spot within a program.

© 2008 Rick Miller — All Rights Reserved

Selection And Iteration Statement Selection Table

The following table provides a quick summary of the C# 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	if (conditional expression) true → statement(s); false → continue processing	Provides an alternative program execution path based on the results of a conditional expression. If the conditional expression evaluates to true its body statements execute. 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	if (conditional expression) true → statement(s); false → else → statement(s); → continue processing	Provides two alternative program execution paths based on the result of a 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.
switch	switch (sbyte, byte, short, ushort, int, uint long, ulong, char, string, or enum-type) { case value : yes → statement(s); break; no; case value : yes → statement(s); break; no; // as many cases as required; default: statement(s); break; } continue processing	The `switch` statement evaluates sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or enum values and executes a matching case and its associated statement block. Use the `break` keyword to exit each `case` statement. Always provide a `default` case.	Use the `switch` statement in place of chained `if/else` statements when you are evaluating sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or enum values.
while	while (conditional expression) true → statement(s); (loop back); false → continue processing	The `while` statement repeatedly executes its statement block based on the results of its conditional expression evaluation. The conditional expression will be evaluated first. If true, the statement body executes and the conditional expression will again be evaluated. If it is false, the statement body is skipped and processing continues as normal.	Use the `while` loop when you want to do something over and over again while some condition is true.

Table 7-1: C# Selection And Iteration Statement Selection Guide

© 2008 Rick Miller — All Rights Reserved

Statement	Execution Diagram	Operation	When To Use
do/while	do { statement(s); } while (conditional expression) false true continue processing	The `do/while` statement operates much like the `while` statement except its body statements are evaluated *at least once* before the conditional expression is evaluated.	Use the `do/while` statement when you want the body statements to be executed at least once.
for	for (initialization expression ; conditional expression ; expression) false true statements(s); continue processing	The `for` statement operates like the `while` statement but offers a more compact way of initializing, comparing, and incrementing counting variables.	Use the `for` statement when you want to iterate over a set of statements for a known number of times.

Table 7-1: C# Selection And Iteration Statement Selection Guide

Summary

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 *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, *string*, or *enum* 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. Implicit case fall-through is only allowed if a case contains no statements.

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-counter variable declaration and initialization, conditional expression evaluation, and loop-counter variable incrementing in a compact format.

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/while` loops. The `continue` statement terminates the current iteration statement loop and forces the start of a new iteration. The `goto` statement is used to jump to any labeled spot within a program.

Skill-Building Exercises

1. **If Statement:** Write a program that reads four string values from the console, converts them into int values using the `Console.ToInt32()` method, and assigns the int values to variables named `int_i`, `int_j`, `int_k`, and

© 2008 Rick Miller — All Rights Reserved

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.

Conditional Expression	Operation
`int_i < int_j`	Print a text message to the console saying that int_i was less than int_j. Use the values of the variables in the message.
`(int_i + int_j) <= int_k`	Print a text message to the console showing the values of all the variables and the results of the addition operation.
`int_k == int_l`	Print a text message to the console saying that int_k was equal to int_l. Use the values of the variables in the text message.
`(int_k != int_i) && (int_j > 25)`	Print a text message to the console that shows the values of the variables.
`(((++int_j) & (--int_l)) > 0)`	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.

Hint: Use the == operator 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 == operator in the following fashion:

```
name_1 == name_2
```

The == operator 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 character 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:

Case	Operation
'A'	Prompt the user to enter two numbers. Add the two numbers the user enters and print the sum to the console.
'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.

© 2008 Rick Miller — All Rights Reserved

Don't forget to use the `break` keyword to exit each `case`. Study the CheckBookBalancer program given in Example 7.14 to see how to read input from the console using the Console.ReadLine() method.

4. **While Statement:** Write a program that prompts the user to enter a letter. Assign the letter to a character 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 C#!"
```

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 integer 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.

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'`.

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 after writing the number of guesses to the console.

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 when the user enters `'N'`.

© 2008 Rick Miller — All Rights Reserved

3. **Simple Calculator:** Write a program that implements a four function calculator that adds, subtracts, multiplies, and divides integer and floating point values.

4. **Calculate Area of Circles:** Write a program that calculates the area of circles using the following formula:

$$A_c = \pi r^2$$

Prompt the user for the value of r. After each iteration ask users if they wish to continue and terminate the program if they enter `'N'`.

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 Adding Machine:** 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 test 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 types can be used as `switch` statement evaluation 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.

© 2008 Rick Miller — All Rights Reserved

7. When would you use a `switch` statement vs. chained `if/else` statements?

8. For what purpose is the `break` keyword used in `switch` statements?

9. What's the effect of using the `break` keyword in an iteration statement?

10. What's the effect of using the `continue` keyword in an iteration statement?

11. What's the purpose of the `goto` statement?

References

Lawrence S. Leff. *Geometry The Easy Way*, Second Edition. Barron's Educational Series, Inc. ISBN: 0-8120-4287-5

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Notes

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Chapter 8

Contax T / Kodak Tri-X

Rick, Kyle, Coralie – Paris, 2005

Arrays

Learning Objectives

- Describe the purpose of an array
- List and describe the use of single and multidimensional arrays
- Describe how array objects are allocated in memory
- Describe the difference between arrays of value types vs. arrays of reference types
- Demonstrate your ability to create arrays using array literals
- 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

 © 2008 Rick Miller — All Rights Reserved

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 C# 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 used to solve many programming problems. 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 simple predefined value 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 associated with array allocation helps 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 enables you to easily understand the concepts behind multidimensional arrays.

Along the way, you will learn the difference between arrays of value 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. I will also explain to you the difference between rectangular and ragged arrays.

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. *Contiguous* means the array elements are located one after the other in memory. *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 types, each element would be the size of an integer and occupy 4 bytes. If, however, an array is declared to contain double 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.

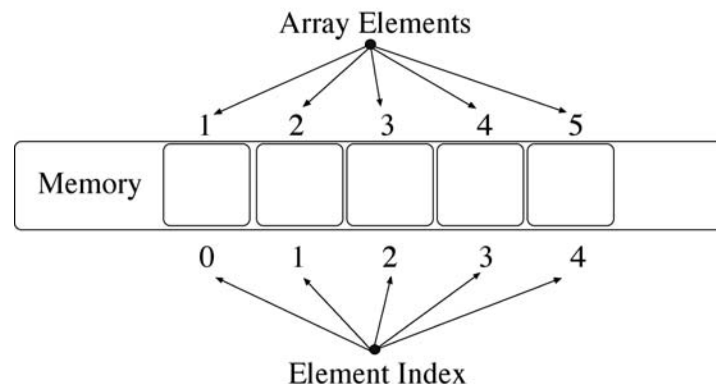


Figure 8-1: Array Elements are Contiguous and Homogeneous

Figure 8-1 shows an array of 5 elements of no specified 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 referenced or accessed by its array index number. An index number is always one less than the element number it

 © 2008 Rick Miller — All Rights Reserved

accesses. For example, when you want to access the 1st element of an array, use index number 0. To access the 2nd element of an array, 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 value 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.

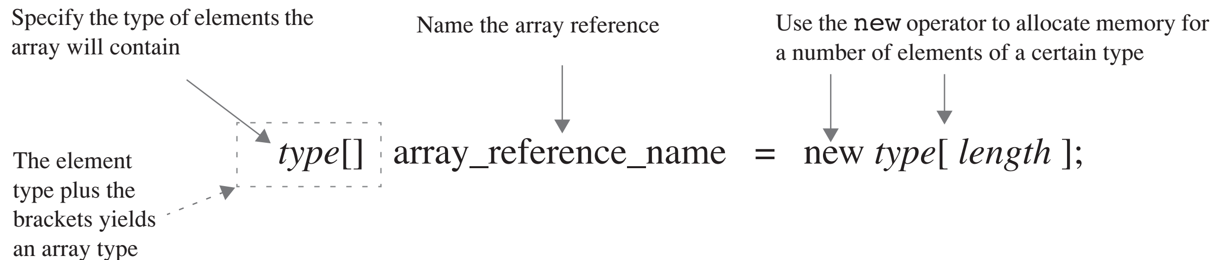


Figure 8-2: Declaring a Single-Dimensional Array

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 value types or reference types. Reference types can include any reference type specified in the .NET API, reference types you create, or third-party types created by someone else.

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* or *rank* you want the array to have. 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 reference to 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 on 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 a length of 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 rectangular array of boolean-type elements and allocate some memory:

```
bool[,] boolean_array_2d = new bool[10,10];
```

The following line of code would create a single-dimensional array of strings:

```
String[] string_array = new String[8];
```

 © 2008 Rick Miller — All Rights Reserved

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*. Access array elements 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.,* the 1st element has index 0, the 2nd element has index 1, ... , the nth element has index n-1)

Functionality Provided By C# Array Types

As you learned in Chapter 6, the C# language has two data-type categories: value types and reference types. Arrays are a special case of reference types. When you create an array in C#, it is an object just like a reference type object. However, C# arrays possess special features over and above ordinary reference types because they inherit from the System.Array class. This section explains what it means to be an array type.

Array-Type Inheritance Hierarchy

When you declare an array in C#, you specify an array type as was shown previously in Figure 8-2. The array you create automatically inherits the functionality provided by the System.Array class, which itself extends from the System.Object class. Figure 8-3 shows the UML inheritance diagram for an array type.

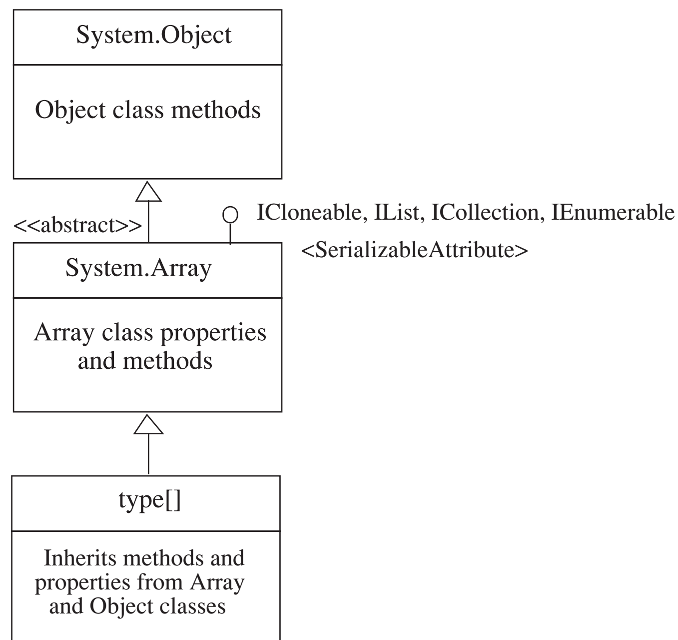


Figure 8-3: Array-Type Inheritance Hierarchy

Referring to Figure 8-3 — the inheritance from the Array and Object classes is taken care of automatically by the C# language when you declare an array. The Array class is a special class in the .NET Framework in that you cannot derive from it directly to create a new array type subclass. Any attempt to explicitly extend from System.Array in your code will cause a compiler error.

 © 2008 Rick Miller — All Rights Reserved

The Array class provides several public properties and methods that make it easy to manipulate arrays. Some of these properties and methods can be accessed via an array reference, while others are meant only to be accessed via the Array class itself. You will see examples of the Array class's methods and properties in action as you progress through this chapter. In the meantime, however, it would be a good idea to access the MSDN website and pay a visit to the System.Array class documentation to learn more of what it has to offer.

Special Properties Of C# Arrays

The Table 8-1 summarizes the special properties of C# 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. However, arrays can be automatically resized with the help of the Array.Resize() method.
Their number of dimensions or rank can be determined by accessing the Rank property.	For example: `int[] int_array = new int[5];` This code declares a single-dimensional array of five integers. The following line of code prints to the console the number of dimensions int_array contains: `Console.WriteLine(int_array.Rank);`
The length of a particular array dimension or rank can be determined via the GetLength() method.	Array objects have a method named GetLength() that returns the value of the length of a particular array dimension or rank. To call the GetLength() method, use the dot operator and the name of the array. For example: `int[] int_array = new int[5];` This code declares and initializes an array of integer elements with length 5. The next line of code prints the length of the int_array to the console: `Console.WriteLine(int_array.GetLength(0));` The GetLength() method is called with an integer argument indicating the desired dimension. In the case of a single-dimensional array, there is only one dimension.
Array bounds are checked by the virtual execution system 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 System.Array class.	Because arrays subclass System.Array they have the functionality of an Array.
Elements are initialized to default values.	Predefined simple value type array elements are initialized to the default value of the particular value type each element is declared to contain. For example, integer array elements are initialized to zero. Each element of an array of references is initialized to null.

Table 8-1: C# Array Properties

Quick Review

C# array types have special functionality because of their special inheritance hierarchy. C# array types directly and automatically inherit the functionality of the System.Array class and implement the ICloneable, IList, ICollection, and IEnumerable interfaces. Arrays are also serializable.

© 2008 Rick Miller — All Rights Reserved

Creating And Using Single-Dimensional Arrays

This section shows you how to declare, create, and use single-dimensional arrays of both value types and reference types. Once you know how a single-dimensional array works, you can easily apply the concepts to multidimensional arrays.

Arrays Of Value Types

The elements of a value type array can be any of the C# predefined value types or value types that you declare (*i.e.*, structures). The predefined value types include *bool*, *byte*, *sbyte*, *char*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *float*, *double*, and *decimal*. 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.cs

```
1    using System;
2
3    public class IntArrayTest {
4      static void Main(){
5        int[] int_array = new int[10];
6        for(int i=0; i<int_array.GetLength(0); i++){
7          Console.Write(int_array[i] + " ");
8         }
9         Console.WriteLine();
10     }
11   }
```

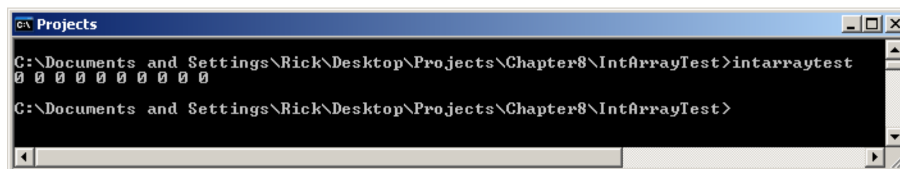


Figure 8-4: Results of Running Example 8.1

Referring to Example 8.1 — this program demonstrates several important concepts. First, an array of integers of length 10 is declared and created on line 5. The name of the array is int_array. To demonstrate that each element of the array is automatically initialized to zero, the `for` statement on line 6 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 Value-Type Array Objects Are Arranged In Memory

Figure 8-5 shows how the integer array int_array declared and created in Example 8.1 is represented in memory. The name of the array, int_array, is a reference to an object in memory of type *System.Int32[]*. The array object is dynamically allocated on the application's memory heap with the `new` operator. 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 property returns the value of the total number of elements in the array, which in this case is 10. The array object's Rank property returns the total number of dimensions in the array, which in this case is 1.

Let's make a few changes to the code given in Example 8.1 by assigning some values to the int_array elements. Example 8.2 adds another `for` loop to the program that initializes each element of int_array to the value of the `for` loop's index variable i.

8.2 IntArrayTest.cs (Mod 1)

 © 2008 Rick Miller — All Rights Reserved

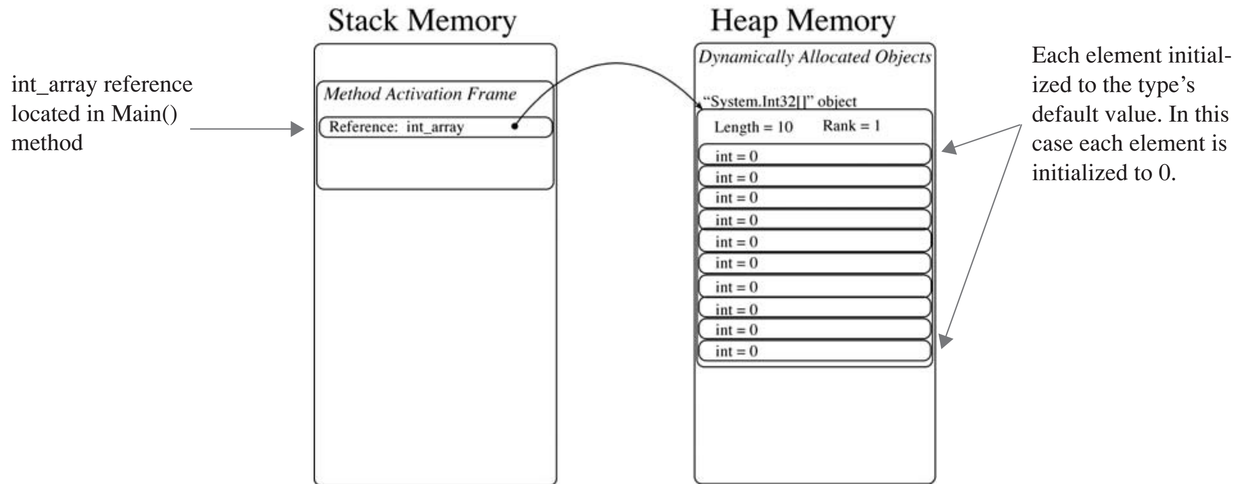


Figure 8-5: Memory Representation of Value Type Array int_array Showing Default Initialization

```
1    using System;
2
3    public class IntArrayTest {
4      static void Main(){
5        int[] int_array = new int[10];
6        for(int i=0; i<int_array.GetLength(0); i++){
7          Console.Write(int_array[i] + " ");
8         }
9        Console.WriteLine();
10       for(int i=0; i<int_array.GetLength(0); i++){
11         int_array[i] = i;
12         Console.Write(int_array[i] + " ");
13        }
14       Console.WriteLine();
15     }
16   }
```

Referring to Example 8.2 — notice on line 11 how the value of the second `for` loop's index variable i is assigned directly to each array element. When the array elements print to the console, each element's value has changed except for the first, which is still zero. Figure 8-6 shows the results of running this program. Figure 8-7 shows the memory representation of int_array after its elements have been assigned their new values.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\IntArrayTestMod1>intarraytest
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\IntArrayTestMod1>
```

Figure 8-6: Results of Running Example 8.2

Finding An Array's Type, Rank, And Total Number of Elements

Study the code shown in Example 8.3, paying particular attention to lines 6 through 10.

8.3 IntArrayTest.cs (Mod 2)

```
1    using System;
2
3    public class IntArrayTest {
4      static void Main(){
5        int[] int_array = new int[10];
6        Console.WriteLine("int_array has rank of " + int_array.Rank);
7        Console.WriteLine("int_array has " + int_array.Length + " total elements");
8        Console.WriteLine("The number of elements in the first (and only) rank is " +
9                   int_array.GetLength(0));
10       Console.WriteLine(int_array.GetType());
11
12       for(int i=0; i<int_array.GetLength(0); i++){
```


© 2008 Rick Miller — All Rights Reserved

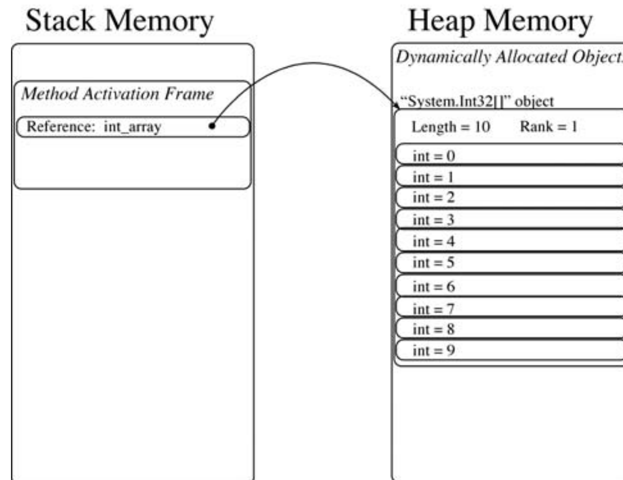


Figure 8-7: Element Values of int_array After Initialization Performed by Second `for` Loop

```
13          Console.Write(int_array[i] + " ");
14        }
15        Console.WriteLine();
16        for(int i=0; i<int_array.GetLength(0); i++){
17          int_array[i] = i;
18          Console.Write(int_array[i] + " ");
19        }
20         Console.WriteLine();
21      }
22    }
```

Referring to Example 8.3 — lines 6 through 10 show how to use Array class methods to get information about an array. On line 6, the Rank property is accessed via the int_array reference to print out the number of int_array's dimensions. On line 7, the Length property returns the total number of array elements. On lines 8 and 9, the GetLength() method is called with an argument of 0 to determine the number of elements in the first rank. In the case of single-dimensional arrays, the Length property and GetLength(0) return the same value. On line 10, the GetType() method determines the type of the int_array reference. It returns the value "System.Int32[]," where the single pair of square brackets signifies an array type. Figure 8-8 gives the results of running this program.

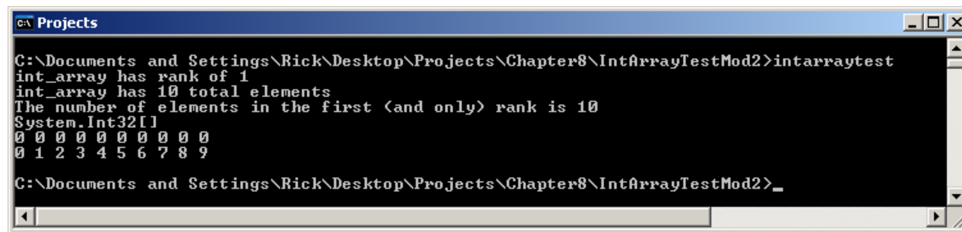


Figure 8-8: Results of Running Example 8.3

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 its 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.

8.4 ArrayLiterals.cs

```
1    using System;
2
3    public class ArrayLiterals {
4      static void Main(){
5        int[] int_array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
6        double[] double_array = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
7
```

 © 2008 Rick Miller — All Rights Reserved

```
8        for(int i = 0; i < int_array.GetLength(0); i++){
9          Console.Write(int_array[i] + " ");
10       }
11       Console.WriteLine();
12       Console.WriteLine(int_array.GetType());
13       Console.WriteLine(int_array.GetType().IsArray);
14
15       Console.WriteLine();
16
17       for(int i = 0; i < double_array.GetLength(0); i++){
18         Console.Write(double_array[i] + " ");
19       }
20       Console.WriteLine();
21       Console.WriteLine(double_array.GetType());
22       Console.WriteLine(double_array.GetType().IsArray);
23     }
24   }
```

Referring to Example 8.4 — the program declares and initializes two arrays using array literal values. On line 5 an array of integers 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 6, an array of doubles named double_array is declared and initialized with double literal values. The contents of both arrays are printed to the console. Array class methods are then used to determine the characteristics of each array and the results are printed to the console. Notice on lines 13 and 22 the use of the IsArray property. It will return true if the reference via which it is called is an array type. Figure 8-9 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\ArrayLiterals>arrayliterals
1 2 3 4 5 6 7 8 9 10
System.Int32[]
True

1 2 3 4 5 6 7 8 9 10
System.Double[]
True

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\ArrayLiterals>_
```

Figure 8-9: Results of Running Example 8.4

Differences Between Arrays Of Value Types And Arrays Of Reference Types

The key difference between arrays of value types and arrays of reference types is that value-type values can be directly assigned to value-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. Instead, each reference element is automatically initialized to *null*. You must explicitly create each object you want each reference element to point to. Alternatively, 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.

8.5 ObjectArray.cs

```
1    using System;
2
3    public class ObjectArray {
4     static void Main(){
5        Object[] object_array = new Object[10];
6        Console.WriteLine("object_array has type " + object_array.GetType());
7        Console.WriteLine("object_array has rank " + object_array.Rank);
8        Console.WriteLine();
9
10       object_array[0] = new Object();
11       Console.WriteLine(object_array[0].GetType());
12       Console.WriteLine();
13
14       object_array[1] = new Object();
15       Console.WriteLine(object_array[1].GetType());
16       Console.WriteLine();
17
18       for(int i = 2; i < object_array.GetLength(0); i++){
19           object_array[i] = new Object();
```


© 2008 Rick Miller — All Rights Reserved

```
20            Console.WriteLine(object_array[i].GetType());
21            Console.WriteLine();
22        }
23      }
24    }
```

Figure 8-10 shows the results of running this program.

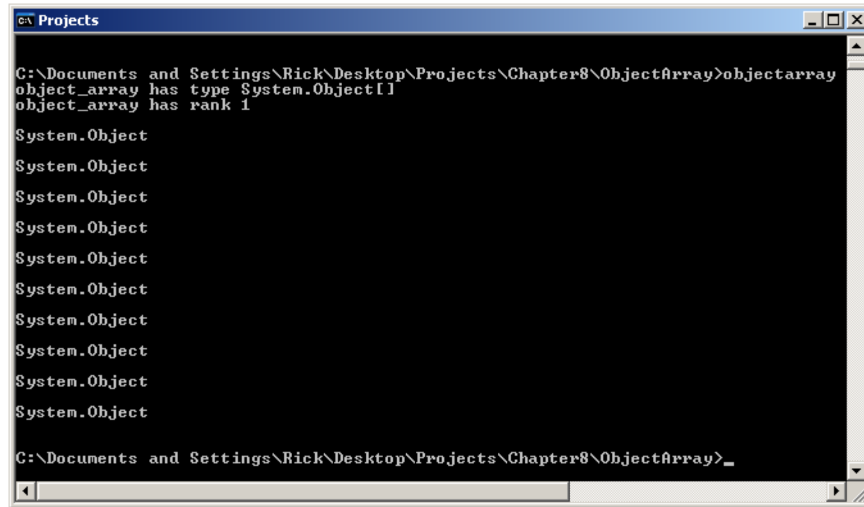


Figure 8-10: Results of Running Example 8.5

Referring to Example 8.5 — on line 5, an array of Objects of length 10 is declared and created. After line 5 executes, the object_array reference points to an array of Objects in memory with each element initialized to null, as is shown in Figure 8-11.

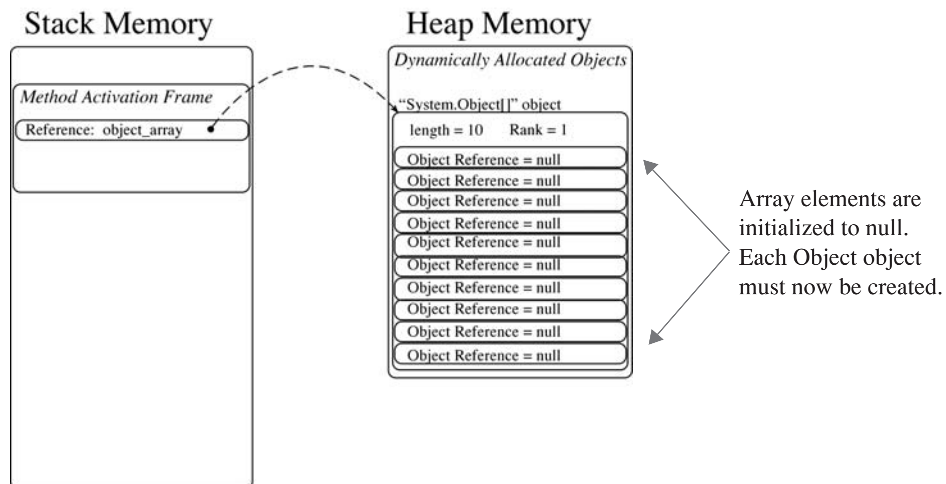


Figure 8-11: State of Affairs After Line 5 of Example 8.5 Executes

On lines 6 and 7, the program writes to the console some information about the object_array, namely, its type and rank. On line 10, a new object of type Object is created and its memory location is assigned to the Object reference located in object_array[0]. The memory picture now looks like that shown in Figure 8-12. Line 11 calls the GetType() method on the object pointed to by object_array[0].

The execution of line 14 results in the creation of another object of type Object in memory. The memory picture now looks like that shown in Figure 8.13. The `for` statement on line 18 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.

© 2008 Rick Miller — All Rights Reserved

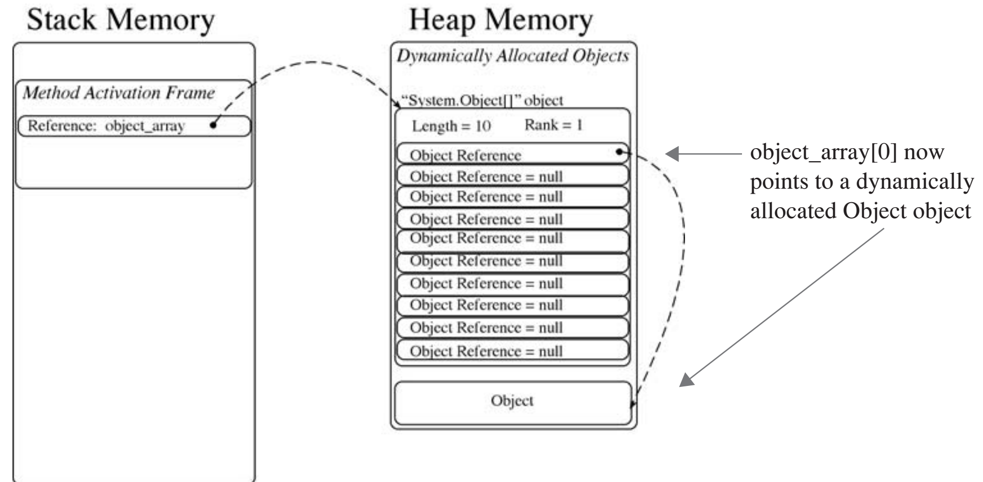


Figure 8-12: State of Affairs After Line 10 of Example 8.5 Executes.

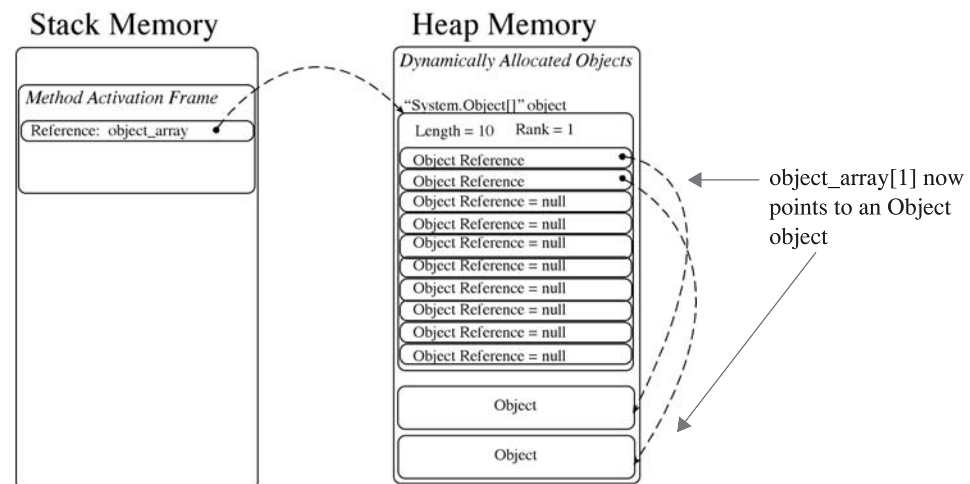


Figure 8-13: State of Affairs After Line 14 of Example 8.5 Executes

Now that you know the difference between value 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.

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.

8.6 MessageArray.cs

```
1    using System;
2
3    public class MessageArray {
4      static void Main(){
5        String name = null;
6        int int_val = 0;
7
```

 © 2008 Rick Miller — All Rights Reserved

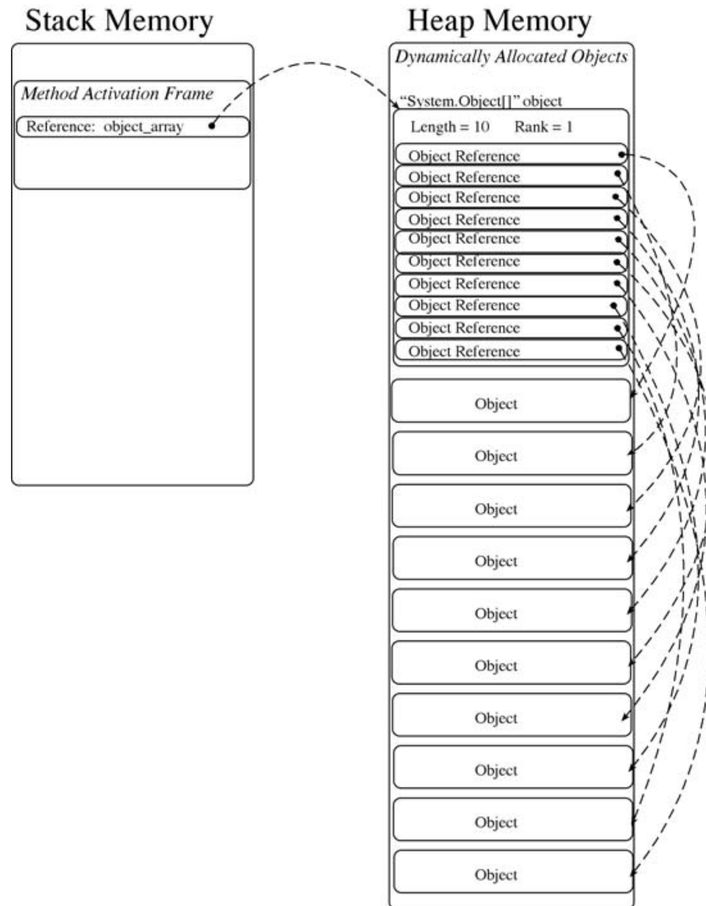


Figure 8-14: Final State of Affairs: All object_array Elements Point to an Object object

```
8        String[] messages = {"Welcome to the Message Array Program",
9                     "Please enter your name: ",
10                    ", please enter an integer: ",
11                    "You did not enter an integer!",
12                    "Thank you for running the Message Array program"};
13
14       const int WELCOME_MESSAGE     = 0;
15       const int ENTER_NAME_MESSAGE  = 1;
16       const int ENTER_INT_MESSAGE   = 2;
17       const int INT_ERROR           = 3;
18       const int THANK_YOU_MESSAGE   = 4;
19
20       Console.WriteLine(messages[WELCOME_MESSAGE]);
21       Console.Write(messages[ENTER_NAME_MESSAGE]);
22       name = Console.ReadLine();
23
24       Console.Write(name + messages[ENTER_INT_MESSAGE]);
25
26       try{
27           int_val = Int32.Parse(Console.ReadLine());
28          }catch(FormatException) { Console.WriteLine(messages[INT_ERROR]); }
29
30       Console.WriteLine(messages[THANK_YOU_MESSAGE]);
31     }
32   }
```

Referring to Example 8.6 — this program creates a single-dimensional array of strings named messages. It initializes each string element using string literals. On lines 14 through 18, an assortment of constants are declared and initialized. These constants are used to index the messages array as is shown on lines 20 and 21. The program simply asks the user to enter a name followed by a request to enter an integer value. If the user fails to enter an integer, the Int32.Parse() method will throw a FormatException. Figure 8-15 shows the results of running this program.

 © 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\MessageArray>messagearray
Welcome to the Message Array Program
Please enter your name: Rick
Rick, please enter an integer: 225
Thank you for running the Message Array program

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\MessageArray>_
```

Figure 8-15: Results of Running Example 8.6

Calculating Averages

The program given in Example 8.7 calculates class grade averages.

8.7 Average.cs

```
1    using System;
2
3    public class Average {
4      static void Main(){
5          double[] grades      = null;
6          double   total       = 0;
7          double   average     = 0;
8          int      grade_count = 0;
9
10         Console.WriteLine("Welcome to Grade Averager");
11         Console.Write("Please enter the number of grades to enter: ");
12         try{
13             grade_count = Int32.Parse(Console.ReadLine());
14            } catch(FormatException) { Console.WriteLine("You did not enter a number!"); }
15
16          if(grade_count > 0){
17              grades = new double[grade_count];
18                 for(int i = 0; i < grade_count; i++){
19                    Console.Write("Enter grade " + (i+1) + ": ");
20                      try{
21                         grades[i] = Double.Parse(Console.ReadLine());
22                      } catch(FormatException) { Console.WriteLine("You did not enter a number!"); }
23                 } //end for
24
25                 for(int i = 0; i < grade_count; i++){
26                    total += grades[i];
27                 } //end for
28
29                average = total/grade_count;
30                Console.WriteLine("Number of grades entered: " + grade_count);
31                Console.WriteLine("Grade average: {0:F2}           ", average);
32
33           }//end if
34      } //end main
35   }// end Average class definition
```

Referring to Example 8.7 — an array reference of doubles named grades is declared on line 5 and initialized to null. On lines 6 through 8, several other program variables are declared and initialized.

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 17 to create the grades array. The program then enters a `for` loop on line 18, reads each grade from the console, converts it to a double, and assigns it to the ith element of the grades array.

After all the grades are entered into the array, the grades are summed in the `for` loop on line 25. The average is calculated on line 29. Notice how numeric formatting is used on line 38 to properly format the double value contained in the average variable. Figure 8-16 shows the results of running this program

Histogram: Letter Frequency Counter

Letter frequency counting is an important part of deciphering messages 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 the letter frequency display to the console. The program ignores all characters except the 26 letters of the alphabet.

8.8 Histogram.cs

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\Averages>average
Welcome to Grade Averager
Please enter the number of grades to enter: 3
Enter grade 1: 89
Enter grade 2: 56.9
Enter grade 3: 98.3
Number of grades entered: 3
Grade average: 81.40

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\Averages>_
```

Figure 8-16: Results of Running Example 8.7

```
1     using System;
2
3     public class Histogram {
4       static void Main(String[] args){
5         int[] letter_frequencies = new int[26];
6         const int A = 0, B = 1, C = 2, D = 3, E = 4, F = 5, G = 6,
7                   H = 7, I = 8, J = 9, K = 10, L = 11, M = 12, N = 13,
8                   O = 14, P = 15, Q = 16, R = 17, S = 18, T = 19, U = 20,
9                   V = 21, W = 22, X = 23, Y = 24, Z = 25;
10        String input_string = null;
11
12        Console.Write("Enter a line of characters: ");
13        input_string = Console.ReadLine().ToUpper();
14
15
16        if(input_string != null){
17          for(int i = 0; i < input_string.Length; i++){
18            switch(input_string[i]){
19             case 'A': letter_frequencies[A]++;
20                       break;
21             case 'B': letter_frequencies[B]++;
22                       break;
23             case 'C': letter_frequencies[C]++;
24                       break;
25             case 'D': letter_frequencies[D]++;
26                       break;
27             case 'E': letter_frequencies[E]++;
28                       break;
29             case 'F': letter_frequencies[F]++;
30                       break;
31             case 'G': letter_frequencies[G]++;
32                       break;
33             case 'H': letter_frequencies[H]++;
34                       break;
35             case 'I': letter_frequencies[I]++;
36                       break;
37             case 'J': letter_frequencies[J]++;
38                       break;
39             case 'K': letter_frequencies[K]++;
40                       break;
41             case 'L': letter_frequencies[L]++;
42                       break;
43             case 'M': letter_frequencies[M]++;
44                       break;
45             case 'N': letter_frequencies[N]++;
46                       break;
47             case 'O': letter_frequencies[O]++;
48                       break;
49             case 'P': letter_frequencies[P]++;
50                       break;
51             case 'Q': letter_frequencies[Q]++;
52                       break;
53             case 'R': letter_frequencies[R]++;
54                       break;
55             case 'S': letter_frequencies[S]++;
56                       break;
57             case 'T': letter_frequencies[T]++;
58                       break;
59             case 'U': letter_frequencies[U]++;
60                       break;
61             case 'V': letter_frequencies[V]++;
62                       break;
63             case 'W': letter_frequencies[W]++;
64                       break;
65             case 'X': letter_frequencies[X]++;
66                       break;
```


© 2008 Rick Miller — All Rights Reserved

```
67          case 'Y': letter_frequencies[Y]++;
68                    break;
69          case 'Z': letter_frequencies[Z]++;
70                    break;
71          default : break;
72         } //end switch
73       } //end for
74
75     for(int i = 0; i < letter_frequencies.Length; i++){
76        Console.Write((char)(i + 65) + ": ");
77        for(int j = 0; j < letter_frequencies[i]; j++){
78         Console.Write('*');
79        } //end for
80        Console.WriteLine();
81     } //end for
82
83    } //end if
84   } // end main
85  } // end Histogram class definition
```

Referring to Example 8.8 — on line 5, an integer array named letter_frequencies is declared and initialized to contain 26 elements, one for each letter of the English alphabet. On lines 6 through 9, 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 10, 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 to upper case using the String.ToUpper() method. Most of the work is done within the body of the `if` statement that starts on line 16. 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 property. Also note that a string's characters can be accessed using array notation. Figure 8-17 gives the results of running this program with a sample line of text.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\HistoGram>histogram
Enter a line of characters: Ohhhhhhhhh I love C Sharp Ohhhhh  yes I do I do I do
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:

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\HistoGram>
```

Figure 8-17: Results of Running Example 8.8

Quick Review

Single-dimensional arrays have one dimension — length. You can get an array's length by calling the GetLength() method with an integer argument indicating the particular dimension in which you are interested. Arrays can have elements of either value or reference types. An array type is created by specifying the type name of array elements followed by one set of brackets, [ ]. Use System.Array class methods and properties to get information about an array.

Each element of an array is accessed via an index value indicated by an integer within a set of brackets (*e.g.*, array_name[0]). Value-type element values can be directly assigned to array elements. When an array of value types

© 2008 Rick Miller — All Rights Reserved

is created, each element is initialized to the type's default value. Each element of an array of references is initialized to null. Each object that a reference element points to must either already exist or be created during program execution.

Creating And Using Multidimensional Arrays

C# supports two kinds of multidimensional arrays: *rectangular* and *ragged*. In this section you will learn how to create and use both kinds of multidimensional arrays. I will also show you how to create multidimensional arrays using the `new` operator as well as how to initialize multidimensional arrays using literal values.

Rectangular Arrays

A *rectangular array* is a multidimensional array whose shape is fixed based on the length of each dimension or rank. All of a rectangular array's dimensions must be specified when the array object is created. Figure 8-18 gives the rectangular array declaration syntax for a two-dimensional array.

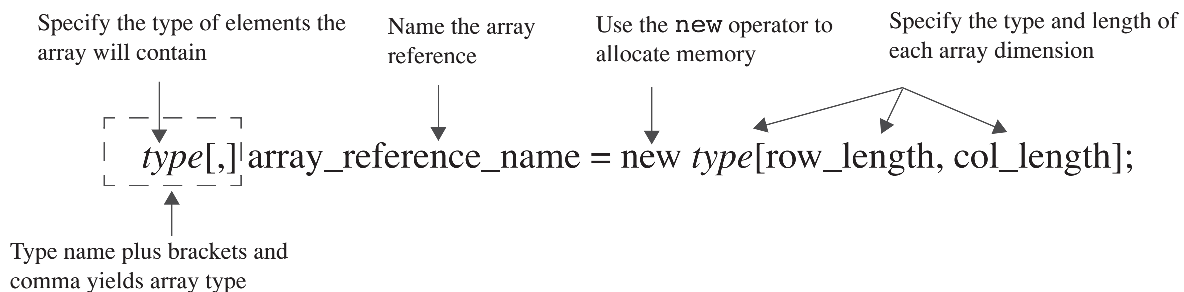


Figure 8-18: Rectangular Array Declaration Syntax

Referring to Figure 8-18 — the type name combined with the brackets and comma yield the array type. For example, the following line of code declares and creates a two-dimensional rectangular array of integers having 10 rows and 10 columns:

```
int[,] int_2d_array = new int[10,10];
```

A two-dimensional array can be visualized as a grid or matrix comprised of rows and columns, as is shown in Figure 8-19. Each element of the array is accessed using two index values, one each for the row and column you wish to access. For example, the following line of code would write to the console the element located in the first row, second column of int_2d_array:

```
Console.WriteLine(int_2d_array[0,1]);
```

Figure 8-19 also includes a few more examples of two-dimensional array element access. Example 8.9 offers a short program that creates a two-dimensional array of integers and prints their values to the console in the shape of a grid.

8.9 TwoDimensionalArray.cs

```
1    using System;
2
3    public class TwoDimensionalArray {
4      static void Main(String[] args){
5
6        try{
7         int rows = Int32.Parse(args[0]);
8         int cols = Int32.Parse(args[1]);
9
10        int[,] int_2d_array = new int[rows, cols];
11        Console.WriteLine("          Array rank: " + int_2d_array.Rank);
12        Console.WriteLine("          Array type: " + int_2d_array.GetType());
13        Console.WriteLine("Total array elements: " + int_2d_array.Length);
14        Console.WriteLine();
15
16        for(int i = 0, element = 1; i<int_2d_array.GetLength(0); i++){
17          for(int  j = 0; j<int_2d_array.GetLength(1); j++){
18           int_2d_array[i,j] = element++;
```


© 2008 Rick Miller — All Rights Reserved

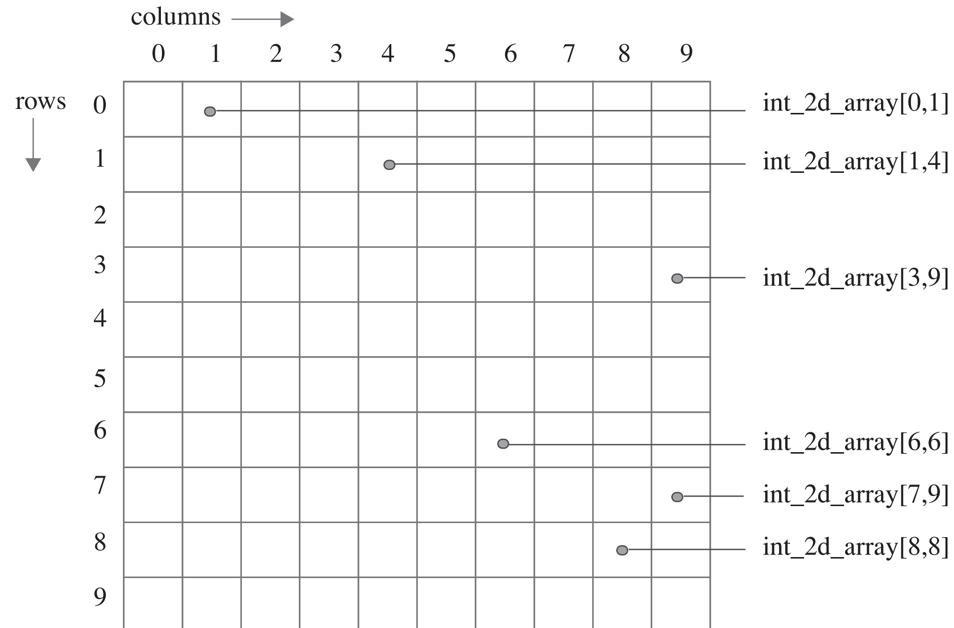


Figure 8-19: Accessing Two-Dimensional Array Elements

```
19            Console.Write("{0:D3} ",int_2d_array[i,j]);
20           }
21            Console.WriteLine();
22         }
23
24        }catch(IndexOutOfRangeException){
25           Console.WriteLine("This program requires two command-line arguments.");
26        }catch(FormatException){
27           Console.WriteLine("Arguments must be integers!");
28        }
29      }
30    }
```

Referring to Example 8.9 — when the program executes, the user enters two integer values on the command line for the desired row and column lengths. These values are read and converted on lines 7 and 8, respectively. The two-dimensional array of integers is created on line 10, followed by several lines of code that writes some information about the array including its rank, type, and total number of elements to the console. The nested `for` statement beginning on line 16 *iterates* over each element of the array. Notice that the outer `for` statement on line 16 declares an extra variable named element. It's used in the body of the inner `for` loop to keep count of how many elements the array contains so that its value can be assigned to each array element. The statement on line 19 prints each array element's value to the console with the help of numeric formatting. Figure 8-20 gives the results of running this program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\RectangularArrays>twodimensionalarray 10 10
          Array rank: 2
          Array type: System.Int32[,]
Total array elements: 100

001 002 003 004 005 006 007 008 009 010
011 012 013 014 015 016 017 018 019 020
021 022 023 024 025 026 027 028 029 030
031 032 033 034 035 036 037 038 039 040
041 042 043 044 045 046 047 048 049 050
051 052 053 054 055 056 057 058 059 060
061 062 063 064 065 066 067 068 069 070
071 072 073 074 075 076 077 078 079 080
081 082 083 084 085 086 087 088 089 090
091 092 093 094 095 096 097 098 099 100

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\RectangularArrays>_
```

Figure 8-20: Results of Running Example 8.9

© 2008 Rick Miller — All Rights Reserved

Initializing Rectangular Arrays With Array Literals

Rectangular arrays can be initialized using literal values in an array initializer expression. Study the code offered in Example 8.10.

8.10 RectangularLiterals.cs

```
1    using System;
2
3    public class RectangularLiterals {
4      static void Main(){
5        char[,] char_2d_array = {{'a', 'b', 'c'},
6                                 {'d', 'e', 'f'},
7                                 {'g', 'h', 'i'}};
8
9        Console.WriteLine("char_2d_array has rank: " + char_2d_array.Rank);
10       Console.WriteLine("char_2d_array has type: " + char_2d_array.GetType());
11       Console.WriteLine("Total number of elements: " + char_2d_array.Length);
12       Console.WriteLine();
13
14       for(int i = 0; i<char_2d_array.GetLength(0); i++){
15         for(int j = 0; j<char_2d_array.GetLength(1); j++){
16           Console.Write(char_2d_array[i,j] + " ");
17         }
18         Console.WriteLine();
19       }
20     }
21   }
```

Referring to Example 8.10 — a two-dimensional array of chars named char_2d_array is declared and initialized on line 5 to have 3 rows and 3 columns. Notice how each row of characters appears in a comma-separated list between a set of braces. Each row of initialization data is itself separated from the next row by a comma, except for the last row of data on line 7. Lines 9 through 11 write some information about the character array to the console, namely, its rank, type, and total number of elements. The nested `for` statement beginning on line 14 iterates over the array and prints each character to the console in the form of a grid. Figure 8-21 shows the results of running this program.

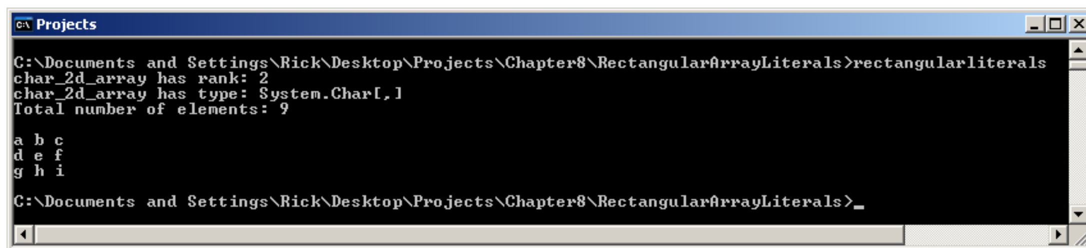


Figure 8-21: Results of Running Example 8.10

Ragged Arrays

A *ragged array* is an array of arrays. Ragged arrays can be any number of dimensions, but the last, or rightmost, dimension is omitted from the array creation expression. Each rightmost array object must then be dynamically created during program execution, resulting in the possibility that the array dimensions may differ in length, hence the name ragged array. Figure 8.22 shows the ragged array declaration syntax for a two-dimensional ragged array. Example 8.11 gives a short program showing the use of a ragged array.

8.11 Ragged2dArray.cs

```
1    using System;
2
3    public class Ragged2dArray {
4      static void Main(){
5        int[][] ragged_2d_array = new int[10][];
6
7        Console.WriteLine("ragged_2d_array has rank: " + ragged_2d_array.Rank);
8        Console.WriteLine("ragged_2d_array has type: " + ragged_2d_array.GetType());
9        Console.WriteLine("Total number of elements: " + ragged_2d_array.Length);
10       Console.WriteLine();
11
12       for(int i = 0; i<ragged_2d_array.GetLength(0); i++){
13         ragged_2d_array[i] = new int[i+1];
14       }
```

 © 2008 Rick Miller — All Rights Reserved

```
15
16      for(int i = 0; i<ragged_2d_array.GetLength(0); i++){
17        for(int j = 0; j<ragged_2d_array[i].GetLength(0); j++){
18          Console.Write(ragged_2d_array[i][j] + " ");
19        }
20        Console.WriteLine();
21      }
22    }
23  }
```

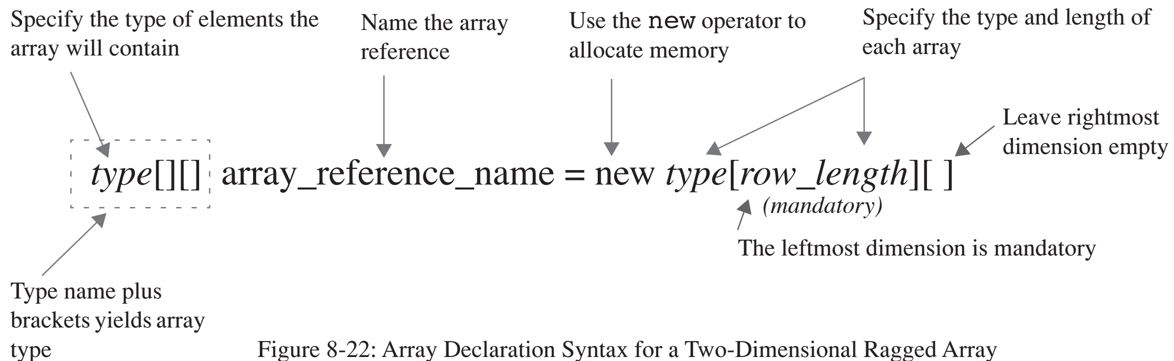


Figure 8-22: Array Declaration Syntax for a Two-Dimensional Ragged Array

Referring to Example 8.11 — on line 5 a two-dimensional ragged array of integers is declared and created. Lines 7 through 9 write some information about the array including its rank, type, and total number of elements to the console. The `for` statement beginning on line 12 creates 10 new arrays of varying lengths and assigns their references to each element of ragged_2d_array. The next `for` statement on line 16 iterates over the ragged two-dimensional array structure and writes the value of each element to the console. Figure 8-23 shows the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\RaggedArray>ragged2darray
ragged_2d_array has rank: 1
ragged_2d_array has type: System.Int32[][]
Total number of elements: 10

0
0 0
0 0 0
0 0 0 0
0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\RaggedArray>
```

Figure 8-23: Results of Running Example 8.11

Multidimensional Arrays In Action

The example presented in this section shows how single and multidimensional arrays can be used together effectively.

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.

8.12 WeightedGradeTool.cs

```
1   using System;
2
3   public class WeightedGradeTool {
4       static void Main() {
5
6           double[,] grades = null;
7           double[] weights = null;
```


© 2008 Rick Miller — All Rights Reserved

```
8           String[] students = null;
9           int student_count = 0;
10          int grade_count = 0;
11          double final_grade = 0;
12
13          Console.WriteLine("Welcome to Weighted Grade Tool");
14
15          /*************** get student count ********************/
16          Console.Write("Please enter the number of students: ");
17          try {
18              student_count = Int32.Parse(Console.ReadLine());
19          }
20          catch (FormatException) {
21              Console.WriteLine("That was not an integer!");
22              Console.WriteLine("Student count will be set to 3.");
23              student_count = 3;
24          }
25
26
27          if (student_count > 0) {
28              students = new String[student_count];
29              /***************** get student names *********************/
30              for (int i = 0; i < students.Length; i++) {
31                  Console.Write("Enter student name: ");
32                  students[i] = Console.ReadLine();
33              }
34
35              /*************** get number of grades per student **********/
36              Console.Write("Please enter the number of grades to average: ");
37              try {
38                  grade_count = Int32.Parse(Console.ReadLine());
39              }
40              catch (FormatException) {
41                  Console.WriteLine("That was not an integer!");
42                  Console.WriteLine("Grade count will be set to 3.");
43                  grade_count = 3;
44              }
45
46              /****************** get raw grades ****************************/
47              grades = new double[student_count, grade_count];
48              for (int i = 0; i < grades.GetLength(0); i++) {
49                  Console.WriteLine("Enter raw grades for " + students[i]);
50                  for (int j = 0; j < grades.GetLength(1); j++) {
51                      Console.Write("Grade " + (j + 1) + ": ");
52                      try {
53                          grades[i, j] = Double.Parse(Console.ReadLine());
54                      }
55                      catch (FormatException) {
56                          Console.WriteLine("That was not a double!");
57                          Console.WriteLine("Grade will be set to 100");
58                          grades[i, j] = 100;
59                      }
60                  }//end inner for
61              }
62
63              /***************** get grade weights ********************/
64              weights = new double[grade_count];
65              Console.WriteLine("Enter grade weights. Make sure they total 100%");
66              for (int i = 0; i < weights.Length; i++) {
67                  Console.Write("Weight for grade " + (i + 1) + ": ");
68                  try {
69                      weights[i] = Double.Parse(Console.ReadLine());
70                  }
71                  catch (FormatException) {
72                      Console.WriteLine("That was not a double!");
73                      Console.WriteLine("The weight will be set to 25");
74                      weights[i] = 25.0;
75                  }
76              }
77
78              /****************** calculate weighted grades *******************/
79              for (int i = 0; i < grades.GetLength(0); i++) {
80                  for (int j = 0; j < grades.GetLength(1); j++) {
81                      grades[i, j] *= weights[j];
82                  }//end inner for
83              }
84
85              /***************** calculate and print final grade ********************/
86              for (int i = 0; i < grades.GetLength(0); i++) {
87                  Console.WriteLine("Weighted grades for " + students[i] + ": ");
88                  final_grade = 0;
```


© 2008 Rick Miller — All Rights Reserved

```
89                  for (int j = 0; j < grades.GetLength(1); j++) {
90                      final_grade += grades[i, j];
91                      Console.Write(grades[i, j] + " ");
92                  }//end inner for
93                  Console.WriteLine(students[i] + "'s final grade is: " + final_grade);
94              }
95          }// end if
96      }// end Main
97  }// end class
```

Figure 8-24 shows the results of running this program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\WeightedGradeTool>weightedgradetool
Welcome to Weighted Grade Tool
Please enter the number of students: 2
Enter student name: Rick
Enter student name: Steve
Please enter the number of grades to average: 4
Enter raw grades for Rick
Grade 1: 77
Grade 2: 87
Grade 3: 99
Grade 4: 66
Enter raw grades for Steve
Grade 1: 23
Grade 2: 44
Grade 3: 78
Grade 4: 98
Enter grade weights. Make sure they total 100%
Weight for grade 1: .25
Weight for grade 2: .25
Weight for grade 3: .25
Weight for grade 4: .25
Weighted grades for Rick:
19.25 21.75 24.75 16.5 Rick's final grade is: 82.25
Weighted grades for Steve:
5.75 11 19.5 24.5 Steve's final grade is: 60.75

C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\WeightedGradeTool>
```

Figure 8-24: Results of Running Example 8.12

Quick Review

C# supports two kinds of multidimensional arrays: *rectangular* and *ragged*. A *rectangular array* is a multidimensional array whose shape is fixed based on the length of each *dimension* or *rank*. All of a rectangular array's dimensions must be specified when the array object is created. A *ragged array* is an array of arrays. Ragged arrays can be any number of dimensions, but the last, or rightmost, dimension is omitted from the array creation expression. Each rightmost array object must then be created during program execution, introducing the possibility that the array's dimensions may differ in length.

The Main() Method's String Array

Now that you have a better understanding of arrays, the Main() method's string array should make 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 C# applications to accept and act upon command-line arguments. The csc 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. This chapter and 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 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.

8.13 CommandLine.cs

© 2008 Rick Miller — All Rights Reserved

```
1    using System;
2    using System.Text;
3
4    public class CommandLine {
5        static void Main(String[] args){
6        StringBuilder sb = null;
7        bool upper_case = false;
8        int start_index = 0;
9
10      /********** check for upper case option **************/
11      if(args.Length > 0){
12         switch(args[0][0]){ // get the first character of the first argument
13           case '-' :
14                    if(args[0].Length > 1){
15                      switch(args[0][1]){ // get the second character of the first argument
16                         case 'U' :
17                         case 'u' : upper_case = true;
18                                    break;
19                         default:   upper_case = false;
20                                    break;
21                      }
22                     }
23                    start_index = 1;
24                    break;
25           default: upper_case = false;
26                    break;
27
28         }// end outer switch
29
30         sb = new StringBuilder();   //create StringBuffer object
31
32      /******* process text string **********************/
33        for(int i = start_index; i < args.Length; i++){
34              sb.Append(args[i] + " ");
35        }//end for
36
37         if(upper_case){
38
39           Console.WriteLine(sb.ToString().ToUpper());
40         }else {
41
42           Console.WriteLine(sb.ToString().ToLower());
43          }//end if/else
44
45      } else { Console.WriteLine("Usage: CommandLine [-U | -u] Text string");}
46
47     }//end main
48   }//end class
```

Figure 8.25 shows the results of running this program.

Figure 8-25: Results of Running Example 8.13

Manipulating Arrays With The System.Array Class

The .NET platform makes it easy to perform common array manipulations such as searching and sorting with the System.Array class. Example 8.14 offers a short program that shows the Array class in action sorting an array of integers.

8.14 ArraySortApp.cs

```
49   using System;
50
51   public class ArraySortApp {
52     static void Main() {
53       int[] int_array = { 100, 45, 9, 1, 34, 22, 6, 4, 3, 2, 99, 66 };
54
55       for (int i = 0; i < int_array.Length; i++) {
```

 © 2008 Rick Miller — All Rights Reserved

```
56            Console.Write(int_array[i] + " ");
57        }
58        Console.WriteLine();
59
60        Array.Sort(int_array);
61
62        for (int i = 0; i < int_array.Length; i++) {
63            Console.Write(int_array[i] + " ");
64        }
65        } // end Main() method
66    } // end ArraySortApp class definition
```

Figure 8-26 shows the results of running this program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\ArraySort>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
C:\Documents and Settings\Rick\Desktop\Projects\Chapter8\ArraySort>_
```

Figure 8-26: Results of Running Example 8.14

Numeric Formatting

C# makes it easy to format numeric strings. You have seen several examples of numeric formatting in both this and the previous chapter. You can format numeric results using the String.Format() method or the Console.Write() or Console.WriteLine() methods.

A format string takes the form $C_f nn$ where C_f is a format specifier character and *nn* specifies the number of decimal digits. Table 8-2 lists the standard C# numeric format strings along with some brief example code.

Character	Description	Example Code	Results
C or c	Currency	Console.Write("{0:C}", 4.5); Console.Write("{0:C}", -4.5);	$4.50 ($4.50)
D or d	Decimal	Console.Write("{0:D5}", 45);	00045
E or e	Scientific	Console.Write("{0:E}", 450000);	4.500000E+005
F or f	Fixed-point	Console.Write("{0:F2}", 45); Console.Write("{0:F0}", 45);	45.00 45
G or g	General	Console.Write("{0:G}", 4.5);	4.5
N or n	Number	Console.Write("{0:N}", 4500000);	4,500,000.00
X or x	Hexadecimal	Console.Write("{0:X}", 450); Console.Write("{0:X}", 0xabcd);	1C2 ABCD

Table 8-2: Numeric Formatting

Summary

C# array types have special functionality because of their special inheritance hierarchy. C# array types directly inherit functionality from the System.Array class and implement the ICloneable, IList, ICollection, and IEnumerable interfaces. Arrays are also serializable.

Single-dimensional arrays have one dimension — length. You can get an array's length by calling the GetLength() method with an integer argument that indicates the dimension in which you are interested. You can also get the length of a single dimensional array by accessing its *Length* property. Arrays can have elements of either value

© 2008 Rick Miller — All Rights Reserved

or reference types. An array type is created by specifying the type name of array elements followed by one set of brackets [ ]. Use System.Array class methods and properties to get information about an array.

Each element of an array is accessed via an index value contained within a set of brackets. Value-type element values can be directly assigned to array elements. When an array of value types is created, each element is initialized to the types default value. Each element of an array of references is initialized to null. Each object that a reference element points to must either already exist or be created during program execution.

C# supports two kinds of multidimensional arrays: rectangular and ragged. A rectangular array is a multidimensional array whose shape is fixed based on the length of each dimension or rank. All of a rectangular array's dimensions must be specified when the array object is created.

A ragged array is an array of arrays. Ragged arrays can be any number of dimensions but the last, or rightmost, dimension is omitted from the array creation expression. Each rightmost array object must then be created during program execution, introducing the possibility that the array's dimensions may differ in length.

Use the built-in methods and properties of the System.Array class to perform certain array manipulations such as sorting.

Skill-Building Exercises

1. **Further Research:** Study the System.Array class and the interfaces it implements to better familiarize yourself with the functionality it provides.

2. **Further Research:** Conduct a web search for different applications for single and multidimensional arrays.

3. **Single-Dimensional Arrays:** Write a program that lets you create a single-dimensional array of integers of different sizes at program runtime using command-line inputs.

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, there are several good sources of information regarding sorting algorithms listed in the references section of this chapter.

6. **Multidimensional Arrays:** Modify Example 8.9 so that it creates two-dimensional arrays of characters. 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.

Suggested Projects

1. **Matrix Multiplication:** Given two matrices A_{ij} and B_{jk}, the product C_{ik} can be calculated with the following equation:

© 2008 Rick Miller — All Rights Reserved

$$C_{ik} = \sum_{j=1}^{n} A_{ij} B_{jk}$$

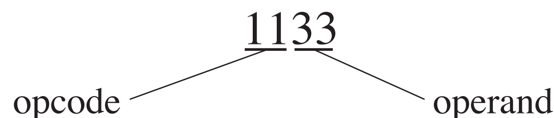
Write a program that multiplies the following matrices together and stores the results in a new matrix. Print the resulting matrix values to the console.

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

2. **Modify Histogram Program:** Modify the histogram program given in Example 8.8 so that it counts the occurrence of the digits 0 through 9 and the punctuation marks period '.', comma ',', question mark '?', colon ':', and semicolon ';'.

3. **Computer Simulator:** You are a C# 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 C# 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, consider 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
Input/Output Operations		
10	READ	Reads an integer value from the console and stores it in memory location identified by the operand.
11	WRITE	Writes the integer value stored in memory location operand to the console.
Load/Store Operations		
20	LOAD	Loads the integer value stored at memory location operand into the accumulator.

Table 8-3: EISCS Machine Instructions

© 2008 Rick Miller — All Rights Reserved

Opcode	Mnemonic	Description
21	STORE	Stores the integer value residing in the accumulator into memory location operand.
Arithmetic Operations		
30	ADD	Adds the integer value located in memory location operand to the value stored in the accumulator and leaves the result in the accumulator.
31	SUB	Subtracts the integer value located in memory location operand from the value stored in the accumulator and leaves the result in the accumulator.
32	MUL	Multiplies the integer value located in memory location operand 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 operand.
Control and Transfer Operations		
40	BRANCH	Unconditional jump to memory location operand.
41	BRANCH_NEG	If accumulator value is less than zero jump to memory location operand.
42	BRANCH_ZERO	If accumulator value is zero then jump to memory location operand.
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 sample 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
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

© 2008 Rick Miller — All Rights Reserved

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 constitute 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 either 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 value types vs. arrays of reference types?

3. C# array types directly inherit functionality from what class?

4. How do you determine the length of an array?

5. (T/F) An array can be resized after it has been created.

6. (T/F) One or more of the dimensions of a rectangular array can be left unspecified upon array object creation.

7. Ragged arrays are _______________ of ______________.

8. When a ragged 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's string array?

© 2008 Rick Miller — All Rights Reserved

References

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Microsoft Developer Network (MSDN) [http://www.msdn.com]

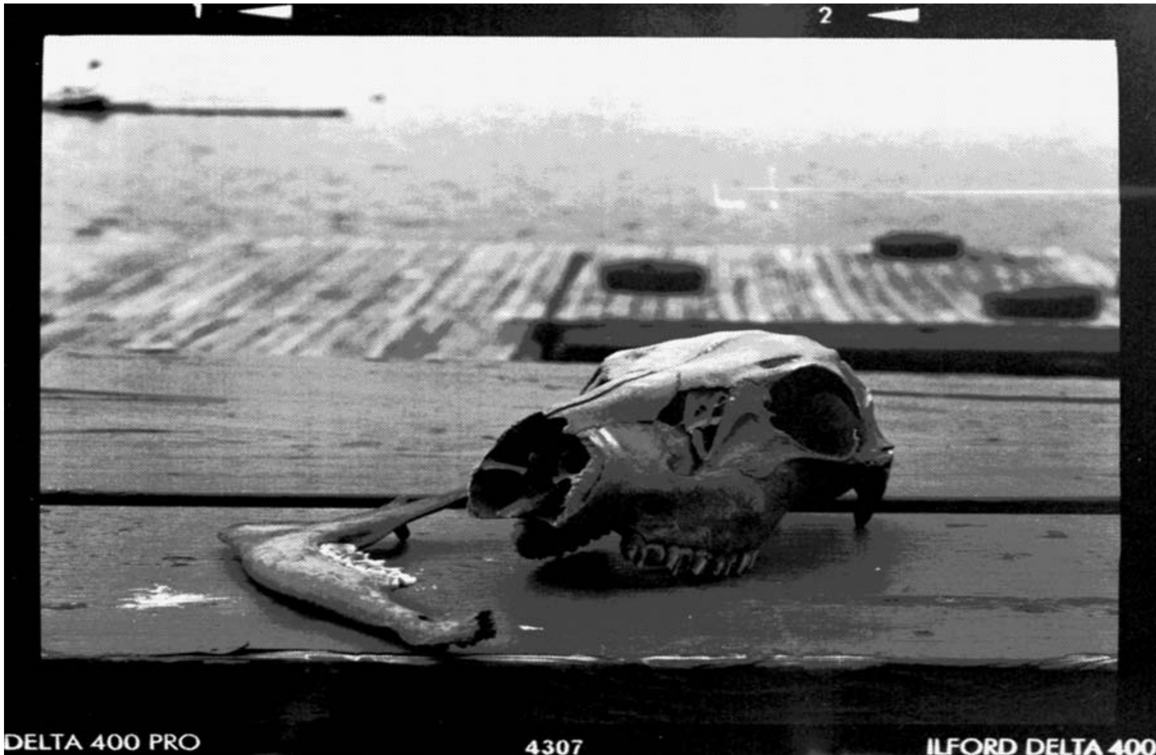
Rick Miller. *Java For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-05-2

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

© 2008 Rick Miller — All Rights Reserved

Chapter 9



Pentax 67 / SMC Takumar 150/2.8 / Ilford Delta 400

Deer Skull

Toward Problem Abstraction: Creating New Data Types

Learning Objectives

- *Describe the purpose and use of abstract data types*
- *State the purpose and use of an enumeration*
- *State the purpose and use of a structure*
- *State the purpose and use of a class*
- *List and describe the differences between structures and classes*
- *State the purpose and use of methods*
- *Demonstrate your ability to create structures, classes, and methods*
- *State the purpose and use of overloaded methods*
- *State the purpose and use of constructor methods*
- *State the definition of the term "method signature"*
- *Demonstrate your ability to overload ordinary methods and constructor methods*
- *State the purpose and use of the keyword "static"*
- *Demonstrate your ability to create class fields and methods*
- *State the purpose and use of a UML class diagram*
- *Demonstrate your ability to create classes that represent abstract data types*

© 2008 Rick Miller — All Rights Reserved

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 the 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 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 provide a measure of code reuse that will save you both work and time.

The primary focus of this chapter is the class construct and its use in abstract data type modeling. At the end of the chapter, I present a brief section on structures and explain the similarities and differences between structures and classes. I then offer suggestions on when you might want to implement an abstract data type as a structure and the ramifications of making such a decision.

The material discussed here builds upon that presented in previous chapters. By now you should be very comfortable using your chosen development environment and a handful of .NET Framework classes. You should be able to create simple C# 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 an expert at looking through the .NET API documentation for classes that can help you solve problems.

Upon completion of this chapter, you will have added several powerful tools to your programmer's toolbag. 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, 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 C#, real number approximations are provided by the float and double 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 which 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 preconceived 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. Successful programmers have 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.

© 2008 Rick Miller — All Rights Reserved

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 must 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 C#, you create a new data type by defining a new enumeration, structure, class, or interface. Arrays and delegates are data types as well, but not useful for the purposes discussed here. These data types can then be used by other data types. This is referred to as design by composition. 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 C# class construct. Most everything you learn about classes also applies to structures. I will discuss the differences between structures and classes at the end of the chapter.

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 ele-
     ments.
   · Create user-defined data types using the class construct
   · Utilize user-defined data types in a C# application
   · Create and manipulate arrays of user-defined data type objects


Tasks:
   · Write a simple program that lets you manage people. The program
     should let users add or delete a person when necessary. The
     program should also let users set and query a person's last,
     middle, and first names as well as his or her birthdate and
     gender. 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

The project specification 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:

© 2008 Rick Miller — All Rights Reserved

- 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 gender
- Set a person's gender
- Query a person's age

The project specification also says that you must store person 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 separate 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 the purpose of that class clear to anyone reading your code.

OK, you have two classes so far: PeopleManager and PeopleManagerApplication. Since you will need person 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 to exceed the project specification if you desire.

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	The Person class will embody the concept of a person entity. A person will have the following attributes: • first name • middle name • last name • gender • birth date The Person class will provide the capability to set and query each of its attributes as well as calculate the age of a person given a person's birth date and the current date.
PeopleManager	The PeopleManager class will manage an array of Person objects. It will have the following attribute: • an array of Person objects The PeopleManager class will also provide the following functionality: • add a person to the array • delete a person from the array • list the people in the array
PeopleManagerApplication	The PeopleManagerApplication class will be the C# application class that has the Main() method. This class will be used to test the functionality of the PeopleManager and Person classes as they are developed.

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 pro-

© 2008 Rick Miller — All Rights Reserved

gram, 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 which piece of its functionality might be provided by a class from the .NET Framework 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 .NET Framework API?" The answer is yes. The place to look for this sort of utility functionality is in the System namespace. There you will find the DateTime class. Take time now to familiarize yourself with the DateTime 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 recommend 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 can be implemented as structures or classes. These structures or 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.

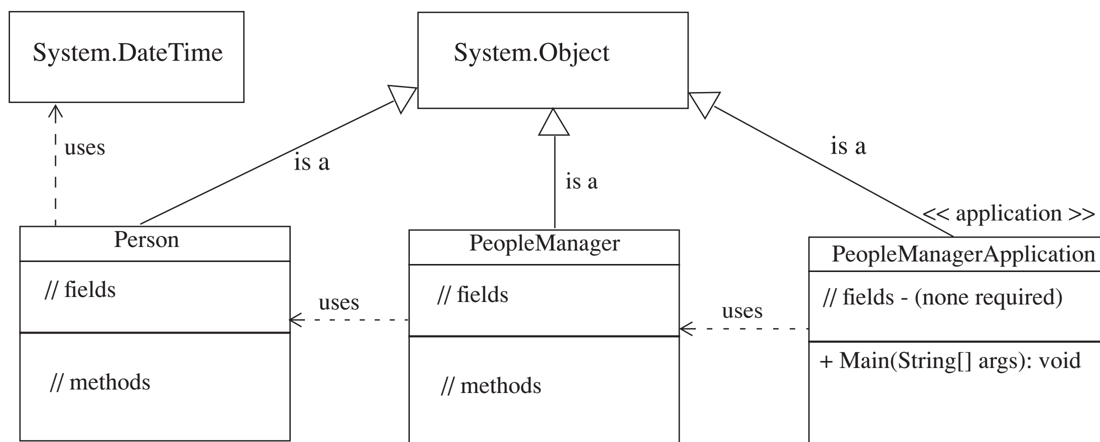


Figure 9-2: Class Diagram for People Manager Classes

© 2008 Rick Miller — All Rights Reserved

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 System.Object class. The Object class serves as the *direct base class* for all reference types that do not explicitly extend another class. I discuss inheritance in detail in Chapter 11.

Each class rectangle can be drawn either 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. This is indicated with the use of the `<<application>>` *stereotype*. A stereotype introduces a new type of element within a system. The name of the new element is contained within the guillemet characters << >>. The application will have one method, Main(). Since it is a class, it could have fields and other methods, 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 fields, properties, and methods as well. These will be developed in the next several sections.

The Person class uses the services of the System.DateTime structure. The DateTime structure 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 discuss the 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 UML class diagram shows the static relationship between classes that participate in a software design. Programmers use the class diagram to express and clarify design concepts to themselves, to other programmers, to management, and to clients.

In UML, a rectangle represents a class. 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.

A stereotype introduces a new type of element within a system. The stereotype name is contained within the guillemet characters << >>.

Generalization and specialization are indicated by 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 Class Construct

This section presents an overview of the C# class construct. You have already been exposed to the structure of a C# application class in Chapter 6 so some of this material will be a review.

Eleven Categories Of Class Members

C# classes can contain eleven different types of members: *fields, constants*, *methods*, *properties*, *events*, *indexers*, *operators*, *instance constructors*, *static constructors, finalizers*, and *nested type declarations*. In this section I

 © 2008 Rick Miller — All Rights Reserved

present a brief description of each member type. The rest of the chapter will demonstrate the use of fields, constants, methods, properties, and instance constructors, as these are the most often used class members. I will discuss the remaining class member types later in the book when their use becomes appropriate. I find it best not to present too much information at one go, or your head will explode!

Fields

Fields are variables that are used to set and maintain object state information. Fields can be either *static* or *non-static*.

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's value exists independently of any particular instance, and therefore does not require a reference to an object to access it. Another term used to describe static fields is *class* or *class-wide fields*.

Non-Static Or Instance Fields

Non-static fields represent attributes for which each object has its 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 birth date. 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.

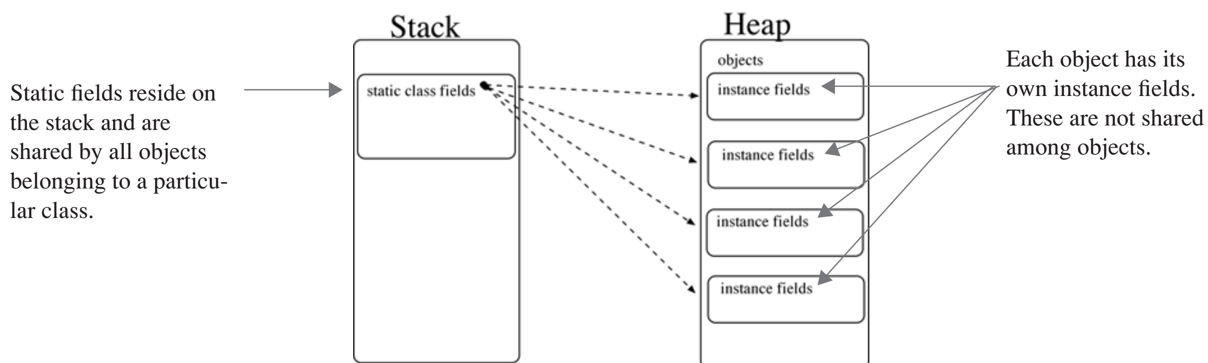


Figure 9-3: Static and Non-Static Fields

Readonly Fields

It's often helpful to have a field maintain its first-assigned value throughout the life of the program. Such a field is said to be a *constant*. Instance readonly fields can be initialized at the point of declaration or in one or more constructors, where each constructor might assign a different value to the readonly field. Static readonly fields can be initialized at the point of declaration or in a static constructor. This can be done by declaring a field to be "readonly" with the `readonly` keyword. Let's take a look at the behavior of an ordinary field vs. a readonly field. Example 9.1 offers a simple code example.

9.1 ReadOnlyTest.cs

```
1    using System;
2
3    public class ReadOnlyTest {
4               int field_1 = 1;
5      readonly int field_2 = 25;
6
7      static void Main(){
8         ReadOnlyTest rot = new ReadOnlyTest();
9         Console.WriteLine(rot.field_1);
10        Console.WriteLine(rot.field_2);
11     }
12   }
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 9.1 — two fields have been declared and initialized on lines 4 and 5. The field named field_2 has been declared readonly. This short program simply prints the field values to the console, as is shown in Figure 9-4.

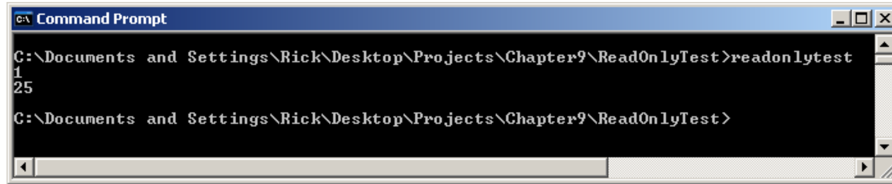


Figure 9-4: Results of Running Example 9.1

As long as you don't try to change the value of a readonly field, you'll be fine. Example 9.2 gives a short program that attempts to change the values of both fields. The error produced when I attempt to compile the program is shown in Figure 9-5.

9.2 ReadOnlyTest.cs (Mod 1)

```
1    using System;
2
3    public class ReadOnlyTest {
4                 int field_1 = 1;
5      readonly int field_2 = 25;
6
7      static void Main(){
8
9         ReadOnlyTest rot = new ReadOnlyTest();
10        Console.WriteLine(rot.field_1);
11        Console.WriteLine(rot.field_2);
12
13        rot.field_1 = 2;
14        rot.field_2 = 26;  // this will cause an error
15
16        Console.WriteLine(rot.field_1);
17        Console.WriteLine(rot.field_2);
18     }
19   }
```

Referring to Example 9.2 — the value of field_1 is modified on line 13 with no problem. The attempt to modify the value of field_2 results in a compiler error.

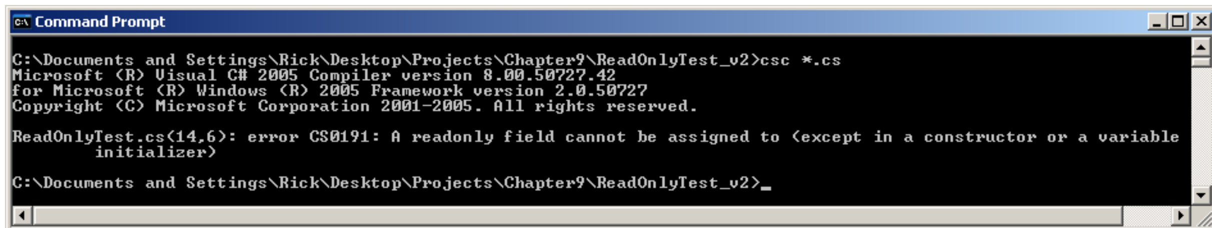


Figure 9-5: Error Resulting from an Attempt to Assign to a Readonly Field

As you can see from the two previous example programs, a readonly field is a constant. Since neither field_1 nor field_2 are declared static, they are both instance fields, which means that each object of type ReadOnlyTest contains its very own copy of these values. It is generally desirable to conserve storage space and declare class constants to be static. This way, the constant values are shared among all objects of a particular class, as Figure 9-3 illustrated. To make field_2 a static field, simply add the keyword `static` to the declaration like so:

```
static readonly int field_2 = 25;
```

Doing this, however, changes the field's behavior. Since it's a static field, it can only be accessed either via the class name or directly from within a class's static or non-static methods, as Example 9.3 illustrates.

9.3 ReadOnlyTest.cs (Mod 2)

```
1    using System;
2
3    public class ReadOnlyTest {
4                       int field_1 = 1;
5      static readonly int field_2 = 25;  // now it's a class-wide constant
6
7      static void Main(){
8         ReadOnlyTest rot = new ReadOnlyTest();
9         Console.WriteLine(rot.field_1);
```

 © 2008 Rick Miller — All Rights Reserved

```
10        Console.WriteLine(ReadOnlyTest.field_2); // access via classname
11        Console.WriteLine(field_2);              // or directly because it is static!
12     }
13   }
```

Referring to Example 9.3 — adding the static keyword to the declaration of field_2 makes it a static field. Static fields can be accessed directly by a class's static or non-static methods or via the class name, as is shown on line 10. Figure 9-6 gives the results of running this program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ReadOnlyTest_mod2>readonlytest
1
25
25

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ReadOnlyTest_mod2>_
```

Figure 9-6: Results of Running Example 9.3

Constants

As you saw in the previous section, class constants can be created by declaring a static readonly field. C# provides a shortcut way to do this with the *const* keyword. Example 9.4 shows how it's done.

9.4 ConstantTest.cs

```
1    using System;
2
3    public class ConstantTest {
4                     int    field_1 = 1;
5      static readonly int    field_2 = 25;
6               const int CONSTANT_1 = 35;
7
8      static void Main(){
9        ConstantTest ct = new ConstantTest();
10       Console.WriteLine(ct.field_1); // can only be accessed via reference since it's non-static
11       Console.WriteLine(ConstantTest.field_2);    // can be accessed via class
12       Console.WriteLine(field_2);                 // or directly because it's static
13       Console.WriteLine(ConstantTest.CONSTANT_1); // can be accessed via class
14       Console.WriteLine(CONSTANT_1);              // or directly because it's static
15     }
16   }
```

Referring to Example 9.4 — on line 6 the keyword `const` declares a class-wide constant member. This is akin to declaring a field to be `static readonly`, but there is a difference, which I explain in the next section. Note that uppercase letters were used to form the constant's identifier to make it stand out in the program. Figure 9-7 gives the results of running this program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ConstantTest>constanttest
1
25
25
35
35

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ConstantTest>_
```

Figure 9-7: Results of Running Example 9,4

The Difference Between `const` and `readonly`; Compile-Time vs. Runtime Constants

A constant declared with the `const` keyword must be initialized at the moment of declaration. The `const` keyword is used to introduce what are called *compile-time* constants. Use the `readonly` keyword to declare a constant if you need to create the constant object using the `new` keyword or if you need to initialize the constant value in a constructor. For example, if you need to create a constant DateTime object that is initialized to a particular date, do something like the following:

```
static readonly DateTime MIN_VALID_SQL_DATE = new DateTime(01, 01, 1753);
```


© 2008 Rick Miller — All Rights Reserved

Properties

A property is a class member that provides access to an object or class attribute. A property provides accessors that contain statements that are executed when its value is read or written. Properties can be static or non-static, read-only, write-only, or read-write.

Properties, at first glance, can be a confusing concept to grasp. One has a tendency to associate properties with fields, but they are more closely related to methods; property accessors get converted into methods during the compilation process.

Instance Properties

An instance property is a non-static member that must be accessed via an object reference.

Static Properties

A static property is a class-wide member that can be accessed via the class name or directly in static and non-static methods.

Read-Only Properties

A read-only property is one whose value can only be read and not written. A read-only property defines a `get` accessor.

Write-Only Properties

A write-only property is one whose value can only be written and not read. A write-only property defines a `set` accessor.

Read-Write Properties

A read-write property is one whose value can be both read and written. A read-write property defines both a `get` and a `set` accessor.

Properties In Action

Example 9.5 gives a short program demonstrating the use of properties.

9.5 PropertiesDemo.cs

```
1    using System;
2
3    public class PropertiesDemo {
4
5      /**** Constants and Fields *****/
6      private const String MESSAGE = "Hello Stranger";
7      private static int field_1 = 1;
8             private int field_2 = 2;
9
10     /***** Properties ******/
11     public String ClassName {
12       get { return this.GetType().ToString(); }
13     }
14
15     public String Message {
16       get { return MESSAGE; }
17     }
18
19     public static int ObjectCount {
20       get { return field_1; }
21       set { field_1 = value; }
22     }
23
24     public int SomeProperty {
25       get { return field_2; }
26       set { field_2 = value; }
27     }
```

 © 2008 Rick Miller — All Rights Reserved

```
28
29     static void Main(){
30       PropertiesDemo pd = new PropertiesDemo();
31       Console.WriteLine(pd.ClassName);
32       Console.WriteLine(pd.Message);
33       Console.WriteLine(ObjectCount);
34       ObjectCount++;
35       Console.WriteLine(ObjectCount);
36       Console.WriteLine(pd.SomeProperty++);
37       Console.WriteLine(pd.SomeProperty);
38     }
39   }
```

Referring to Example 9.5 — the PropertiesDemo class has one constant and two fields. One of the fields, field_1, is a static field. I have defined four properties. Note that each property has a type and a name. Property names are by convention formed with *camel case*. Camel case means the first letter of each word in the identifier name is uppercase and the remaining letters of each word are lowercase.

Each property's accessor definitions are enclosed in the property's body, which is denoted by the opening and closing brace. A read-only property has a `get` accessor defined, which itself has an opening and closing brace and can contain any number of statements as long as it eventually returns an object of the property's specified type. For example, the ClassName property whose definition begins on line 11 is a read-only property. It defines a `get` accessor that returns a string value. Note that the ClassName property computes the value of the string, in this case the class name, by making a series of method calls on the appropriate objects. Compare the behavior of the ClassName property to that of the Message property whose definition starts on line 15. The Message property is a read-only property that simply returns the value of the MESSAGE constant.

The ObjectCount property is a read-write property because it defines both get and set accessors. It is also a static property because its definition includes the use of the `static` keyword. Note on line 21 the use of the implicit parameter named "value" in the set accessor. Remember that these accessors will be ultimately invoked as method calls. The value parameter is automatically supplied by the compiler when a set accessor is called.

The SomeProperty property is a read-write instance property.

These four properties are used in the body of the Main() method that starts on line 29. Note specifically how properties can be used in ways similar to fields. Instance properties must be accessed in a static method via an object reference. Static properties can be accessed via the class name or directly in static and instance methods. Figure 9-8 gives the results of running Example 9.5.

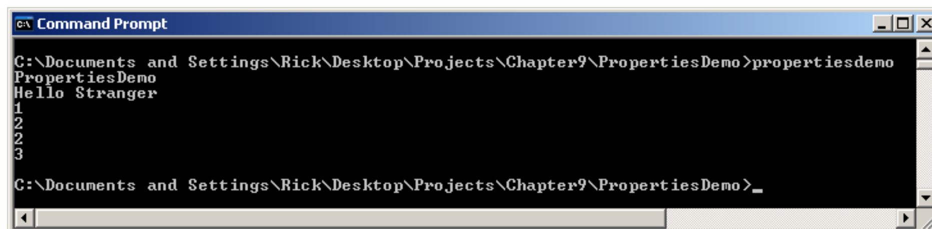


Figure 9-8: Results of Running Example 9.5

Methods

A method is a class member that implements a series of instructions that can be executed or *called* via a class or object. Methods, like fields and properties, can be static or non-static.

Methods can share the same name as long as their method signatures differ. This is referred to as method *overloading*. A method's signature includes its name, and the number and type of its formal parameters. I cover methods in greater detail in this chapter in the Methods section.

Instance Constructors

An *instance constructor*, or simply a *constructor*, is a special type of method that contains the instructions required to properly initialize an object. A constructor method takes the same exact name as the class in which it appears and has no return type.

 © 2008 Rick Miller — All Rights Reserved

A *default constructor* is a constructor that has an empty parameter list. (*i.e.,* It takes no parameters.) If you fail to define a default constructor, the compiler will generate one for you.

Constructors, like ordinary methods, can be overloaded. This comes in handy when you want to define several different ways to create an object.

Constructors are usually declared to have public accessibility, although in some cases it's helpful to declare them to be protected or private so you can maintain full control over how and when an instance object is created. (Refer to the Singleton pattern in Chapter 25 for an example.)

Use instance constructors to initialize non-static readonly fields. This is especially helpful if you needed the read-only constant value to be initialized differently according to which constructor was called.

You'll see many examples of instance constructors throughout this book.

Static Constructors

A static constructor is a special method that contains the instructions required to properly initialize static class fields. Static constructors take no parameters and are called automatically by the runtime environment when the program executes. The use of access modifiers is not allowed with static constructors. (*i.e.*, A static constructor cannot be public or private.) Use a static constructor if you need to initialize static readonly constants.

Events

An event is a class member that enables a class or an object to provide notifications. I cover events in detail in *Chapter 12 — Windows Forms Programming*, and *Chapter 13 — Custom Events*.

Operators

An operator is a member that defines what it means to apply certain expression operators (*like the '+' or '==' operators for example*) to objects. This *operator overloading* allows you to create well-behaved objects. Operator overloading is covered in detail in *Chapter 21 — Operator Overloading*.

Indexers

An indexer is a class member that allows an object to be indexed like an array. I give an example of an indexer in *Chapter 14- Collections*.

Nested Type Declarations

A type definition that appears within the body of a class is referred to as a *nested type*. The most often seen example of this is when an enumerated type (*enumeration*) is declared within the body of a class. The overuse of nested types leads to hard-to-read and hard-to-maintain code. Generally avoid using them unless you have a compelling reason to do so.

Finalizers

A *finalizer* is a class member that's called automatically when an object is collected by the runtime environment garbage collector. A finalizer contains the instructions required to clean up the object. An example of object clean-up might be the release of network resources or file handles used during the object's lifetime.

A finalizer method takes the same name of the class with the tilde character '~' prepended to its name. Finalizers take no parameters and cannot be called explicitly. Because a finalizer method cannot be called explicitly, there is no telling when it will be called. Therefore, the release of critical resources should not be left, as a rule, to the whims of the garbage collector. In practice, use ordinary methods that can be called explicitly to provide critical object clean-up services. The finalizer can then be relied upon as a back-up.

© 2008 Rick Miller — All Rights Reserved

Access Modifiers

Use the access modifiers `public`, `protected`, `private`, `internal`, and `protected internal` to control access to class members. If no access is specified, then private is assumed. The following sections describe the use of these access modifiers in greater detail.

Public

The keyword `public` indicates that the member is accessible to all client code. Generally speaking, most of the methods, constants, and properties you declare in a class will be public.

Private

The keyword `private` indicates that the member 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, though if you're not careful, you can breach this *encapsulation* by absentmindedly returning a reference to a private field via a method or property.

You can also declare methods 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's public interface.

Protected

The keyword `protected` prevents horizontal access to members, but allows them to be inherited or accessed by subclasses. I discuss the protected keyword in detail in *Chapter 11 — Inheritance*.

Internal

The meaning of the `internal` keyword is "for use within this program". An alternative meaning for the internal keyword might be "local public". Essentially, if you declare a member to have internal accessibility, it can be freely accessed by other classes and members within the same assembly. This includes separate .netmodules that are compiled together using the `/addmodule` compiler switch.

If, however, you create a dynamic link library (dll) using the `/target:library` compiler switch, then internal members are accessible to other internal classes and members within that dll, but are not available for use by external code that links to it.

Protected Internal

A member declared to have `protected internal` accessibility is visible to all components contained within the dll just as the internal keyword specifies. Additionally, it can be inherited by subclasses of the member's containing type regardless of the assembly to which the subclass belongs.

The Concepts Of Horizontal Access, Interface, and Encapsulation

The term *horizontal access* describes the access a client object has to the members 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 members via the class name, or 2) by creating an instance of the class and accessing its public non-static members via an object reference.

The members (usually constants, properties, methods, constructors, and events) a class exposes as public are collectively referred to as its *public interface*. Client code becomes dependent upon these public interface members. The wrong kind of change to a class's interface will break any code that depends upon that interface. When changing a class's public interface, the rule-of-thumb is that you can add public members but never remove them. If you look through the .NET API you will see lots of classes with *deprecated* members. A deprecated member is a member that

© 2008 Rick Miller — All Rights Reserved

is targeted for deletion in some future version of the API. These members are not yet removed because doing so would break existing programs that use (depend upon) those members.

Any member declared private is said to be encapsulated within its class, as it is 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, programmers may think up a new ways to enhance 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 9-9 illustrates the concept of horizontal access and the effects of using `public` and `private` access modifiers.

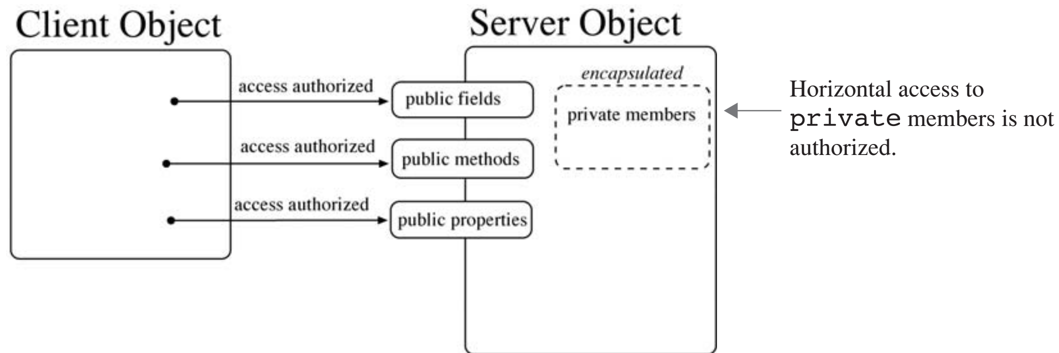


Figure 9-9: Horizontal Access Controlled via Access Modifiers public and private

The concepts of public interfaces, 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 concepts every time you write code in the C# language.

Quick Review

C# classes can contain eleven different types of members: *fields, constants*, *methods*, *properties*, *events*, *indexers*, *operators*, *instance constructors*, *static constructors, finalizers*, and *nested type declarations*.

The access modifiers `public`, `protected`, `private`, `internal`, and `protected internal` are used to control access to class and instance members. If no access is specified then private is assumed.

The term *horizontal access* describes the access a client object has to the members 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 members via the class name, or 2) by creating an instance of the class and accessing its public non-static members via an object reference.

Methods

A method is a named module of executable program functionality. A method contains program statements that, when grouped together, represent a basic level of code reuse. 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 C# language, a method must belong to a class; methods cannot exist or be defined outside of a class construct.

© 2008 Rick Miller — All Rights Reserved

Method Naming: Use Action Words That Indicate The Method's Purpose

Use action words (verbs) when naming a method that provide an indication of the method's intended purpose. See *Appendix C: Identifier Naming and Self-Commenting Code* for a detailed discussion on how to formulate identifier names in a way that makes your code humanly readable.

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*. Your 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 subsection 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 pulls 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, and parameter list. I discuss these method components in detail below. Figure 9-10 shows the structure of a method definition.

```
method_modifiers_opt return_type or void_opt method_name( parameter_list_opt ){
        // method body - program statements go here
  }
```

Figure 9-10: Method Definition Structure

Any piece of the method definition structure shown in Figure 9-10 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 focus on just a few of the potentially many method variations you can write. You will be gradually introduced to different method variations 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 C# 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.e.*, grants horizontal access to the method).

Table 9-2: Method Modifiers

© 2008 Rick Miller — All Rights Reserved

Modifier	Description
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.
internal	The meaning of the `internal` keyword is "for use within this program". Internal methods can be freely accessed by other classes and members within the same assembly. This includes separate .netmodules compiled together using the `/addmodule` compiler switch. If, however, you create a dynamic link library (dll) using the `/target:library` compiler switch, then internal methods are accessible to other internal classes and members within that dll, but are not available for use by external code that links to it.
protected internal	A method declared to have `protected internal` accessibility is visible to all components contained within the assembly as specified by the *internal* keyword, and for use (*i.e.,* can be inherited) by subclasses of the member's containing type, regardless to which assembly the subclass belongs.
static	The `static` keyword declares a static or class method.
abstract	The `abstract` keyword declares a method that contains no body (no implementation). 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 — Inheritance*.
new	The `new` keyword declares a member with the same name or method signature as an inherited member. This new member hides the base member. The `new` keyword is covered in detail in *Chapter 11 — Inheritance*.
override	The `override` keyword designates a method as overriding an inherited method. The `override` keyword is covered in detail in *Chapter 11 — Inheritance*.
virtual	The `virtual` keyword designates a method as being virtual. A virtual method can be overridden in a subclass. The `virtual` keyword is covered in detail in *Chapter 11 — Inheritance*.
sealed	The `sealed` keyword prevents a method from being overridden in derived classes. Sealed methods are covered in detail in *Chapter 11 — Inheritance*.
extern	The `extern` keyword designates a method as being external. An external method is one that is implemented in a language other than C#. (C++ or C for example) The use of the `extern` keyword is not covered in this book.

Table 9-2: Method Modifiers

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 later in this section.

© 2008 Rick Miller — All Rights Reserved

Method Name (mandatory)

The method name is mandatory. As I 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. As a result, it 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, a name, and an optional modifier (`ref` or `out`). The name of the parameter has local scope within the body of the method and hides any field members having the same name. By default, arguments are passed to method parameters by value. The `ref` parameter modifier can be used to pass arguments by reference. The `out` parameter modifier is used to return values to the calling program via the method arguments.

Method Body (optional for abstract or external 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 external method, omit the braces and terminate the method declaration with a semicolon.

Method Definition Examples

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 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.

© 2008 Rick Miller — All Rights Reserved

Method Signatures

Methods have a distinguishing characteristic known as a signature. A method's signature consists of its name and the number, modifiers, and types of its parameters. Method modifiers and return types are not part of a method's signature. It's important that you understand the concept of method signatures 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 chose not to define a constructor method for a particular class if you desire. In that case, the compiler creates 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 set up or build an object when it's created in memory. If you do not provide one, the compiler will create a default constructor for you, but it may or may not provide the level of functionality you require.

 © 2008 Rick Miller — All Rights Reserved

Building And Testing The Person Class

Now that you have been introduced to the C# 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, properties, and methods. As you build the Person class, you may discover that you need to add members as necessary to fully implement the class to your satisfaction. That's a normal part of the design and 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.cs source file and the Person class definition shell. Example 9.6 gives the code for this early stage of the program.

9.6 Person.cs (1st Iteration)

```
1    public class Person {
2
3    } //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. Because you are defining a class that contains no Main() method, you'll get an error stating such unless you create a module using the `/target:module` compiler switch. The complete command required to compile the Person.cs file will be:

```
csc /target:module Person.cs
```

A successful compilation results in no errors or warnings, and a file named Person.netmodule will be written to the project directory.

The next thing to do is 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

After consulting Table 9-1, you learn that the Person class represents a person entity in our problem domain. Each person has his or her own name, gender, and birth date, so these are good candidates for instance fields in the Person class. Example 9.7 shows the Person class code after the instance fields have been added.

9.7 Person.cs (2nd Iteration)

```
1    using System;
2
3    public class Person {
4      private String   _firstName;
5      private String   _middleName;
6      private String   _lastName;
7      private String   _gender;
8      private DateTime _birthday;
9
10   } // end Person class
```

Two .NET API classes are used for the fields in Example 9.7: String and DateTime. The using keyword on line 1 provides shortcut naming for both the String and DateTime types. 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 properties or methods. Let's define a few of those right now. But, before you move on, compile the Person.cs source file again to make sure you didn't break anything. Compiling the Person class in its present state will result in several compiler warnings, one for each unused field. You may safely ignore those warnings for now.

© 2008 Rick Miller — All Rights Reserved

Defining Person Properties And Constructor Method

Now that several Person class instance fields have been created, it's time to define a way to set and manipulate those fields. My approach starts by defining Person's instance properties. I will then use the properties in one or more constructor methods to set the value of each field at the time of object creation. After you've defined several properties and a constructor method, you can use them to test those aspects of Person class behavior.

Adding Properties

Properties are the preferred way to get or set an object's attributes. In this simple example, the initial set of properties defined for the Person class will correspond to its instance fields. Example 9.8 shows the code for an initial set of read/write properties.

9.8 Person.cs (3rd Iteration)

```
1    using System;
2
3    public class Person {
4
5      // private instance fields
6      private String   _firstName;
7      private String   _middleName;
8      private String   _lastName;
9      private String   _gender;
10     private DateTime _birthday;
11
12
13     // public properties
14     public String FirstName {
15       get { return _firstName; }
16       set { _firstName = value; }
17     }
18
19     public String MiddleName {
20       get { return _middleName; }
21       set { _middleName = value; }
22     }
23
24     public String LastName {
25       get { return _lastName; }
26       set { _lastName = value; }
27     }
28
29     public String Gender {
30       get { return _gender; }
31       set { _gender = value; }
32     }
33
34     public DateTime BirthDay {
35       get { return _birthday; }
36       set { _birthday = value; }
37     }
38
39    } // end Person class
```

Referring to Example 9.8 — each read/write property implements a get and set accessor. Remember that the identifier named "value" is an implied parameter. Compiling the Person.cs file in its current state will clear up the unused field warnings now that each field is used in a property definition.

Adding A Constructor Method

The purpose of a 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 matching the fields contained in the Person class. These parameters will then be used to initialize each field. The approach I will take will be to initialize the fields via the properties. Example 9.9 gives the code for the Person class definition after the constructor has been added.

 © 2008 Rick Miller — All Rights Reserved

9.9 Person.cs (4th Iteration)

```
1    using System;
2
3    public class Person {
4
5      // private instance fields
6      private String   _firstName;
7      private String   _middleName;
8      private String   _lastName;
9      private String   _gender;
10     private DateTime _birthday;
11
12     // constructor
13     public Person(String firstName, String middleName, String lastName,
14              String gender, DateTime birthday){
15        FirstName = firstName;
16        MiddleName = middleName;
17        LastName = lastName;
18        Gender = gender;
19        BirthDay = birthday;
20     }
21
22     // public properties
23     public String FirstName {
24       get { return _firstName; }
25       set { _firstName = value; }
26     }
27
28     public String MiddleName {
29       get { return _middleName; }
30       set { _middleName = value; }
31     }
32
33     public String LastName {
34       get { return _lastName; }
35       set { _lastName = value; }
36     }
37
38     public String Gender {
39       get { return _gender; }
40       set { _gender = value; }
41     }
42
43     public DateTime BirthDay {
44       get { return _birthday; }
45       set { _birthday = value; }
46     }
47    } // end Person class
```

Referring to Example 9.9 — the Person constructor method begins on line 13. Notice that it is declared to be public and has no return value. It has five parameters. Each parameter is used in the body of the constructor to set the Person's properties. The properties, in turn, set the values of their corresponding fields.

OK, now that you've got the constructor written, compile the Person.cs source file to ensure you didn't break anything. It's now time to test this puppy.

Testing The Person Class: A Miniature Test Plan

Testing the Person class at this stage of the development cycle consists of creating a Person object and then writing and reading each of its properties. When you create a Person object using the constructor defined in the previous section, you are testing that constructor. The constructor method exercises each property's set accessor. Printing the value of each property to the console would test each property's get accessor.

Use The PeopleManagerApplication Class As A Test Driver

To test the Person class functionality you'll need to create an application class. Since you need to create the PeopleManagerApplication class anyway, you may 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 9.10 gives the code for the PeopleManagerApplication class with a few lines of code that tests the functionality of the Person class developed thus far.

 © 2008 Rick Miller — All Rights Reserved

9.10 PeopleManagerApplication.cs (Testing Person)

```
1    using System;
2
3    public class PeopleManagerApplication {
4      public static void Main(){
5        Person p1 = new Person("Ulysses", "S", "Grant", "Male", new DateTime(1822, 04, 22));
6        Console.WriteLine(p1.FirstName + " " + p1.MiddleName + " " + p1.LastName + " "
7                          + p1.Gender + " " + p1.BirthDay);
8      } // end Main
9    } // end class definition
```

Referring to Example 9.10 — notice how a new DateTime object must be created before being used as an argument for the Person constructor method. To compile this program with the Person.netmodule, use the following command:

```
csc /addmodule:Person.netmodule PeopleManagerApplication.cs
```

Figure 9-11 shows the results of running this program. Everything appears to run fine. It's now time to add a few more features to the Person class.

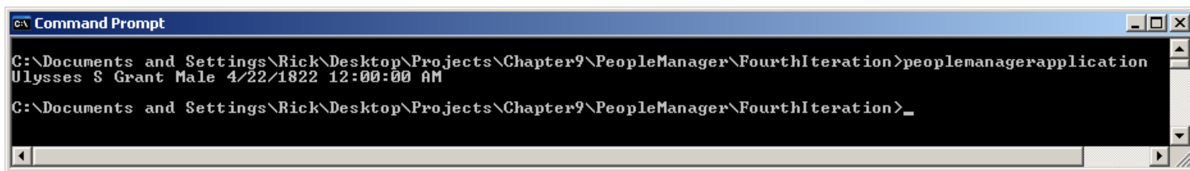


Figure 9-11: Results of Running Example 9.10

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. Think for a moment how you might go about doing this in real life. You might ask people for their birth date and perform the calculation yourself, or you could just ask them how old they are and let them do the calculation for you. I will take the later approach. I'll add a read-only property named Age that computes a Person object's age and returns the result. Example 9.11 shows the modified Person class code.

9.11 Person.cs (5th Iteration)

```
1    using System;
2
3    public class Person {
4
5      // private instance fields
6      private String   _firstName;
7      private String   _middleName;
8      private String   _lastName;
9      private String   _gender;
10     private DateTime _birthday;
11
12     public Person(String firstName, String middleName, String lastName,
13             String gender, DateTime birthday){
14       FirstName = firstName;
15       MiddleName = middleName;
16       LastName = lastName;
17       Gender = gender;
18       BirthDay = birthday;
19     }
20
21     // public properties
22     public String FirstName {
23       get { return _firstName; }
24       set { _firstName = value; }
25     }
26
27     public String MiddleName {
28       get { return _middleName; }
29       set { _middleName = value; }
30     }
31
32     public String LastName {
33       get { return _lastName; }
34       set { _lastName = value; }
35     }
36
```

 © 2008 Rick Miller — All Rights Reserved

```
37     public String Gender {
38       get { return _gender; }
39       set { _gender = value; }
40     }
41
42     public DateTime BirthDay {
43       get { return _birthday; }
44       set { _birthday = value; }
45     }
46
47      public int Age {
48        get {
49          int years = DateTime.Now.Year - _birthday.Year;
50          int adjustment = 0;
51            if(DateTime.Now.Month < _birthday.Month){
52              adjustment = 1;
53            }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
54                      adjustment = 1;
55                  }
56          return years - adjustment;
57        }
58      }
59
60   } // end Person class
```

Referring to Example 9.11 — the Age property definition begins on line 47. As you can see, calculating someone's age takes some doing.

After making the necessary modifications to the Person class you can test the changes in the PeopleManagerApplication class. Example 9.12 shows the code for the modified PeopleManagerApplication class. Figure 9-12 shows the results of running this program.

9.12 PeopleManagerApplication.cs (Testing Person Age property)

```
1    using System;
2
3    public class PeopleManagerApplication {
4      public static void Main(){
5        Person p1 = new Person("Ulysses", "S", "Grant", "Male", new DateTime(1822, 04, 22));
6        Console.WriteLine(p1.FirstName + " " + p1.MiddleName + " " + p1.LastName + " "
7                          + p1.Gender + " " + p1.BirthDay);
8        Console.WriteLine(p1.FirstName + " is " + p1.Age + " years old!");
9      } // end Main
10   } // end class definition
```

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\PeopleManager\FifthIteration>peoplemanagerapplication
Ulysses S Grant Male 4/22/1822 12:00:00 AM
Ulysses is 185 years old!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\PeopleManager\FifthIteration>
```

Figure 9-12: Results of Running Example 9.12

Adding Features To The Person Class: Convenience Properties

The Age property seems to work pretty well. However, it's a hassle to get a Person object's full name and other vital information by calling each individual property. It might be a good idea to add a property that will do the job for you. While you're at it, you could add a property that returns both the full name and age. Each of these properties can use the services of existing properties. Example 9.13 shows the modified Person class.

9.13 Person.cs (6th Iteration)

```
1    using System;
2
3    public class Person {
4
5      // private instance fields
6      private String   _firstName;
7      private String   _middleName;
8      private String   _lastName;
9      private String   _gender;
10     private DateTime _birthday;
11
12     public Person(String firstName, String middleName, String lastName,
```


© 2008 Rick Miller — All Rights Reserved

```
13              String gender, DateTime birthday){
14         FirstName = firstName;
15         MiddleName = middleName;
16         LastName = lastName;
17         Gender = gender;
18         BirthDay = birthday;
19       }
20
21       // public properties
22       public String FirstName {
23         get { return _firstName; }
24         set { _firstName = value; }
25       }
26
27       public String MiddleName {
28         get { return _middleName; }
29         set { _middleName = value; }
30       }
31
32       public String LastName {
33         get { return _lastName; }
34         set { _lastName = value; }
35       }
36
37       public String Gender {
38         get { return _gender; }
39         set { _gender = value; }
40       }
41
42       public DateTime BirthDay {
43         get { return _birthday; }
44         set { _birthday = value; }
45       }
46
47        public int Age {
48          get {
49            int years = DateTime.Now.Year - _birthday.Year;
50            int adjustment = 0;
51              if(DateTime.Now.Month < _birthday.Month){
52                adjustment = 1;
53              }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
54                        adjustment = 1;
55                      }
56            return years - adjustment;
57          }
58        }
59
60       public String FullName {
61         get { return FirstName + " " + MiddleName + " " + LastName; }
62       }
63
64       public String FullNameAndAge {
65         get { return FullName + " " + Age; }
66       }
67
68     } // end Person class
```

Referring to Example 9.13 — the FullName property appears on line 60. It concatenates the FirstName, MiddleName, and LastName properties and returns the resulting String object that represents the Person object's full name.

The FullNameAndAge property on line 64 utilizes the services of the FullName and Age properties. This is a good example of code reuse at the class level. Since the properties 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 9.14 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 9-13 shows the results of running this program.

9.14 PeopleManagerApplication.cs
(Testing Person FullNameAndAge Property)

```
1     using System;
2
3     public class PeopleManagerApplication {
4       public static void Main(){
5         Person p1 = new Person("Ulysses", "S", "Grant", "Male", new DateTime(1822, 04, 22));
6         Console.WriteLine(p1.FullNameAndAge);
7       }
8     }
```

 © 2008 Rick Miller — All Rights Reserved

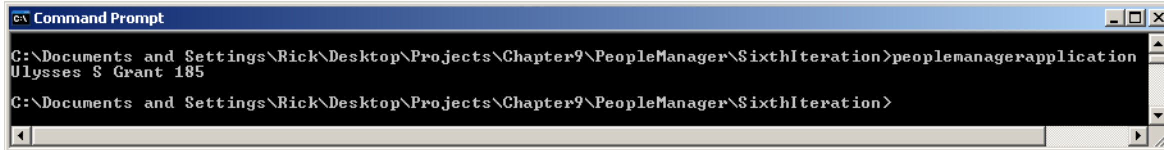


Figure 9-13: Results of Running Example 9.14

Adding Features To The Person Class: Finishing Touches

It's time to take a step back and look at the Person class with an eye towards adding any methods, properties, or other members that might make its usage easier or more intuitive. There are, in fact, many ways to improve upon the design of Person class, but some of what can be done will have to wait until you've gone a little farther in the book.

One aspect of performance we can address here is the addition of a default constructor. Because up to this point I have failed to implement a default constructor, Person objects can be created with meaningless values assigned to each field. To prevent this from happening, add a private default constructor. This will force the use of the one public constructor currently supported by the Person class.

Another helpful member to add is an overriding ToString() method. Although I do not formally cover the concept of method overriding until *Chapter 11 — Inheritance*, it won't hurt to give you a peek at a simple example.

One last thing. It would be nice to limit the range of authorized values the Gender property can assume. This is a perfect use for an enumeration. Example 9.15 gives the code for the improved Person class.

9.15 Person.cs (7th Iteration)

```
1     using System;
2
3     public class Person {
4
5       //enumeration
6       public enum Sex {MALE, FEMALE};
7
8       // private instance fields
9       private String   _firstName;
10      private String   _middleName;
11      private String   _lastName;
12      private Sex      _gender;
13      private DateTime _birthday;
14
15      //private default constructor
16      private Person(){}
17
18      public Person(String firstName, String middleName, String lastName,
19                    Sex gender, DateTime birthday){
20         FirstName = firstName;
21         MiddleName = middleName;
22         LastName = lastName;
23         Gender = gender;
24         BirthDay = birthday;
25      }
26
27      // public properties
28      public String FirstName {
29        get { return _firstName; }
30        set { _firstName = value; }
31      }
32
33      public String MiddleName {
34        get { return _middleName; }
35        set { _middleName = value; }
36      }
37
38      public String LastName {
39        get { return _lastName; }
40        set { _lastName = value; }
41      }
42
43      public Sex Gender {
44        get { return _gender; }
45        set { _gender = value; }
46      }
47
48      public DateTime BirthDay {
```

 © 2008 Rick Miller — All Rights Reserved

```
49      get { return _birthday; }
50      set { _birthday = value; }
51    }
52
53     public int Age {
54       get {
55         int years = DateTime.Now.Year - _birthday.Year;
56         int adjustment = 0;
57           if(DateTime.Now.Month < _birthday.Month){
58             adjustment = 1;
59           }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
60                   adjustment = 1;
61                 }
62         return years - adjustment;
63       }
64     }
65
66     public String FullName {
67       get { return FirstName + " " + MiddleName + " " + LastName; }
68     }
69
70    public String FullNameAndAge {
71      get { return FullName + " " + Age; }
72    }
73
74    public override String ToString(){
75      return FullName + " is a " + Gender + " who is " + Age + " years old.";
76    }
77
78  } // end Person class
```

Referring to Example 9.15 — the enumeration Sex is defined on line 6 and provides two authorized values: MALE and FEMALE. The _gender field's type on line 12 is now Sex vs. String. A similar change was made to the constructor's gender parameter. The private default constructor appears on line 17. On line 43, the type of the Gender property was changed to Sex. Finally, the overriding ToString() method definition begins on line 74.

Example 9.16 shows the modified Person class being tested in the PeopleManagerApplication class.

9.16 PeopleManagerApplication.cs
(Testing modified Person class)

```
1   using System;
2
3   public class PeopleManagerApplication {
4     public static void Main(){
5       Person p1 = new Person("Ulysses", "S", "Grant", Person.Sex.MALE, new DateTime(1822, 04, 22));
6       Console.WriteLine(p1);
7     } // end Main
8   } // end class definition
```

Referring to Example 9.16 — note the use of the enumeration in the constructor argument list to set the Person object's gender. Also note now that because Object's ToString() method has been overridden, all that's required to print a Person object's vital information is to simply call the WriteLine() method with the argument p1. Figure 9-14 shows the results of running this program.

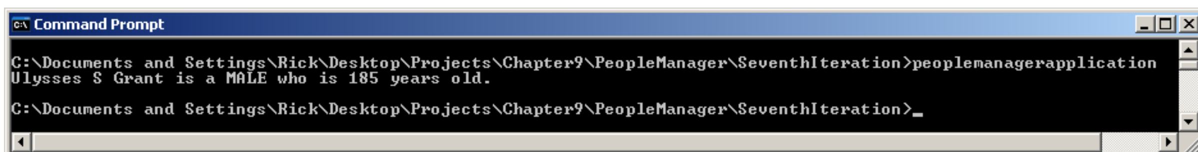


Figure 9-14: Results of Running Example 9.16

Quick Review

Incrementally build and test abstract data types by iteratively applying the steps of the development cycle. Start with the class definition shell and then add fields, properties, and methods as required to fulfill the class's design objectives.

Test class functionality with the help of a test driver. A test driver is a small program that's used to exercise the functionality of another program.

 © 2008 Rick Miller — All Rights Reserved

Building and Testing The PeopleManager Class

Now that the Person class is finished, it's time to shift focus to the PeopleManager class. Consulting Table 9-1 again reveals that the PeopleManager class will manipulate an array of Person objects. It must 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.

The same approach used to develop the Person class is used here to develop the PeopleManager class. The development cycle is applied iteratively to yield the final result.

Defining The PeopleManager Class Shell

Example 9.17 gives the source code for the PeopleManager class definition shell. Compile the code the same way you did the Person class by using the compiler's `/target:module` switch.

9.17 PeopleManager.cs (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. Example 9.18 gives the modified source code for the PeopleManager class after the declaration of a Person array named people_array.

9.18 PeopleManager.cs (2nd Iteration)

```
1    public class PeopleManager {
2      Person[] people_array;
3
4
5    } // end PeopleManager class
```

Additional fields may be required, but for now this is a good start. You can compile this file in its current state using two approaches. If you want to use the Person.netmodule created earlier you can use the following compiler command:

```
csc /target:module /addmodule:Person.netmodule PeopleManager.cs
```

Alternatively, you can compile both the Person.cs and PeopleManager.cs files together using the following compiler command:

```
csc /target:module Person.cs PeopleManager.cs
```

Both approaches yield a new module named PeopleManager.netmodule. Now it's time to add some methods.

Defining PeopleManager Constructor Methods

We'll give the PeopleManager class two constructors. One will be a default constructor, but unlike the Person's private default constructor, this one will be public for the reasons you'll soon see. The other constructor will do most of the dirty work of initializing the PeopleManager object. This includes initializing the people_array and any other fields we add to the PeopleManager class. The default constructor will simply call the other constructor with a default array length value.

To create the people_array object, you will need to know how long the array must be. (*i.e.*, how many Person references you need it to store.) 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

 © 2008 Rick Miller — All Rights Reserved

with some default length value. In the following example, I use a default length of 10. Example 9.19 gives the modified PeopleManager class after the constructors have been added.

9.19 PeopleManager.cs (3rd Iteration)

```
1    using System;
2
3    public class PeopleManager {
4      // private fields
5      private Person[] people_array;
6
7      // overloaded constructor
8      public PeopleManager(int length){
9        people_array = new Person[length];
10     }
11
12     // default constructor
13     public PeopleManager():this(10){ }
14
15   } // end PeopleManager class
```

Referring to Example 9.19 — the constructor on line 8 takes an integer parameter named length and uses it to dynamically create the people_array in memory. The default constructor starts on line 13. It takes no parameters. It calls the constructor defined on line 8 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. Then add some more methods 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 9.20 gives the source code for the PeopleManager class containing the newly created AddPerson() and ListPeople() methods.

9.20 PeopleManager.cs (4th Iteration)

```
1    using System;
2
3    public class PeopleManager {
4      // private fields
5      private Person[] people_array;
6      int index = 0;
7
8      // overloaded constructor
9      public PeopleManager(int length){
10       people_array = new Person[length];
11     }
12
13     // default constructor
14     public PeopleManager():this(10){ }
15
16
17     public void AddPerson(String firstName, String middleName, String lastName,
18                           Person.Sex gender, int dob_year, int dob_month, int dob_day){
19       if(index >= people_array.Length){
20          index = 0;
21       }
22       if(people_array[index] == null){
23         people_array[index++] = new Person(firstName, middleName, lastName, gender,
24                                            new DateTime(dob_year, dob_month, dob_day));
25       }
26     } // end method
27
28
29     public void ListPeople(){
30       for(int i = 0; i<people_array.Length; i++){
31         if(people_array[i] != null){
32           Console.WriteLine(people_array[i]);
33         }
34       }
35     } // end method
36   } // end PeopleManager class
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 9.20 — 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 firstName, middleName, lastName, gender, dob_year, dob_month, and dob_day. The first thing the AddPerson() method does is to check to see if the value of the index field is greater than or equal to the length of the people_array. If so, it resets its value to 0. This guards against the possibility of exceeding the bounds of the array. Next, the AddPerson() method checks 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.

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. If the array element is not null (meaning it points to a Person object) it uses that array element as an argument to the WriteLine() method, which in turn calls the Person object's ToString() method automatically.

Testing The PeopleManager Class

You can use the PeopleManagerApplication class once again to test the functionality of the PeopleManager class. Example 9.21 gives the source code for the modified PeopleManagerApplication class. Figure 9-15 shows the results of running this program.

9.21 PeopleManagerApplication.cs
(Testing the PeopleManager class)

```
1    using System;
2
3    public class PeopleManagerApplication {
4      public static void Main(){
5       PeopleManager pm = new PeopleManager(); // default constructor call
6       pm.AddPerson("Jeff", "J", "Meyer", Person.Sex.MALE, 1975, 03, 12);
7       pm.AddPerson("Pete", "M", "Luongo", Person.Sex.MALE, 1967, 06, 18);
8       pm.AddPerson("Alex", "T", "Remily", Person.Sex.MALE, 1965, 11, 24);
9       pm.ListPeople();
10     } // end Main
11   } // end class definition
```

Referring to Example 9.21 — the PeopleManager default constructor is tested on line 5. This also tests the other PeopleManager constructor. Killed two birds with one stone here! The AddPerson() method is tested on lines 6 through 8, 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 as well.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\PeopleManager\EightIteration>peoplemanagerapplication
Jeff J Meyer is a MALE who is 32 years old.
Pete M Luongo is a MALE who is 40 years old.
Alex T Remily is a MALE who is 42 years old.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\PeopleManager\EightIteration>_
```

Figure 9-15: Results of Running Example 9.21

Adding Features To The PeopleManager Class

The PeopleManager class now implements two out of three required features. You can add Person objects to the people_array and you can list information about each Person object 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. 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 a specific element within the array. A good candidate name for such a method might be InsertPersonAtIndex(). Example 9.22 gives the source code for the modified PeopleManager class.

 © 2008 Rick Miller — All Rights Reserved

9.22 PeopleManagerClass.cs (5th Iteration)

```
1    using System;
2
3    public class PeopleManager {
4      // private fields
5      private Person[] people_array;
6      int index = 0;
7
8      // overloaded constructor
9      public PeopleManager(int length){
10       people_array = new Person[length];
11     }
12
13     // default constructor
14     public PeopleManager():this(10){ }
15
16     public void AddPerson(String firstName, String middleName, String lastName,
17                           Person.Sex gender, int dob_year, int dob_month, int dob_day){
18       if(index >= people_array.Length){
19          index = 0;
20       }
21       if(people_array[index] == null){
22         people_array[index++] = new Person(firstName, middleName, lastName, gender,
23                                            new DateTime(dob_year, dob_month, dob_day));
24       }
25     } // end method
26
27     public void ListPeople(){
28       for(int i = 0; i<people_array.Length; i++){
29         if(people_array[i] != null){
30           Console.WriteLine(people_array[i]);
31         }
32       }
33     } // end method
34
35
36     public void DeletePersonAtIndex(int index){
37       if(!(index < 0) || (index >= people_array.Length)){
38         people_array[index] = null;
39         this.index = index;
40       }
41     }
42
43     public void InsertPersonAtIndex( int index, String firstName, String middleName,
44                                      String lastName, Person.Sex gender, int dob_year,
45                                      int dob_month, int dob_day){
46         if(!(index < 0) || (index >= people_array.Length)){
47           this.index = index;
48           people_array[this.index++] = new Person(firstName, middleName, lastName, gender,
49                                              new DateTime(dob_year, dob_month, dob_day));
50         }
51      }
52
53   } // end PeopleManager class
```

Referring to Example 9.22 — examine closely for a moment the DeletePersonAtIndex() method whose definition starts on line 36. 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 `if` statement on line 37 enforces 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. A similar test is made on the index parameter of the InsertPersonAtIndex().

Example 9.23 gives the source code for the PeopleManagerApplication class that tests the newly added PeopleManager class functionality. Figure 9-16 shows the results of running this program.

9.23 PeopleManagerApplication.cs

```
1    using System;
2
3    public class PeopleManagerApplication {
4      public static void Main(){
5       PeopleManager pm = new PeopleManager(); // default constructor call
6       pm.AddPerson("Jeff", "J", "Meyer", Person.Sex.MALE, 1975, 03, 12);
7       pm.AddPerson("Pete", "M", "Luongo", Person.Sex.MALE, 1967, 06, 18);
8       pm.AddPerson("Alex", "T", "Remily", Person.Sex.MALE, 1965, 11, 24);
9       pm.ListPeople();
10      Console.WriteLine("----------------------------------------");
11      pm.DeletePersonAtIndex(0);
12      pm.ListPeople();
13      Console.WriteLine("----------------------------------------");
```

 © 2008 Rick Miller — All Rights Reserved

```
14      pm.InsertPersonAtIndex(0, "Coralie", "S", "Miller", Person.Sex.FEMALE, 1963, 04, 04);
15      pm.ListPeople();
16     } // end Main
17   } // end class definition
```

```
Command Prompt
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\PeopleManager\NinthIteration>peoplemanagerapplication
Jeff J Meyer is a MALE who is 32 years old.
Pete M Luongo is a MALE who is 40 years old.
Alex T Remily is a MALE who is 42 years old.
----------------------------------------
Pete M Luongo is a MALE who is 40 years old.
Alex T Remily is a MALE who is 42 years old.
----------------------------------------
Coralie S Miller is a FEMALE who is 44 years old.
Pete M Luongo is a MALE who is 40 years old.
Alex T Remily is a MALE who is 42 years old.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\PeopleManager\NinthIteration>
```

Figure 9-16: Results of Running Example 9.23

Quick Review

The PeopleManager class implementation process followed the same pattern as that of class Person. It started with the class shell and added fields and methods as required to implement the necessary functionality. Develop code incrementally by applying the development cycle in an iterative fashion.

More About Methods

In this section I'd like to focus your attention on several behavioral aspects of methods you will find helpful to fully understand before attempting more complex programming projects. You need to know the difference between *value parameters* and *reference parameters*, and be aware of local variable scoping rules.

Value Parameters And Reference Parameters

There are two ways to pass arguments to methods: 1) by *value*, or 2) by *reference*. A method parameter that omits the optional `ref` modifier is a value parameter by default. It's critical that you understand completely the difference between these two modes of parameter behavior or your methods may not work as you expect.

Value Parameters: The Default Parameter Passing Mode

Two sorts of things can be passed as arguments to a method: 1) a value type object or 2) a reference that points to an object, otherwise simply referred to as a reference. When an argument is passed to a method, a *copy* of the argument is made and assigned to its associated method parameter. This is referred to as *pass by copy* or *pass by value*. I say again, behind the scenes, both categories of objects, value and reference, are copied into memory areas accessible to the method. Once copied, value types behave one way and reference types behave another way.

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 value type int parameter and one Object reference parameter. This means that SomeMethod() can take two arguments: the first must be an integer and the second can be a reference to any Object. Remember — classes are reference types and structures are value types! 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

© 2008 Rick Miller — All Rights Reserved

stages of the call to SomeMethod(). When SomeMethod() executes, it is only operating on its parameters, meaning it is only operating on copies of the original argument values.

For value 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 as long as 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 9-17 illustrates these concepts using a class's fields as method arguments.

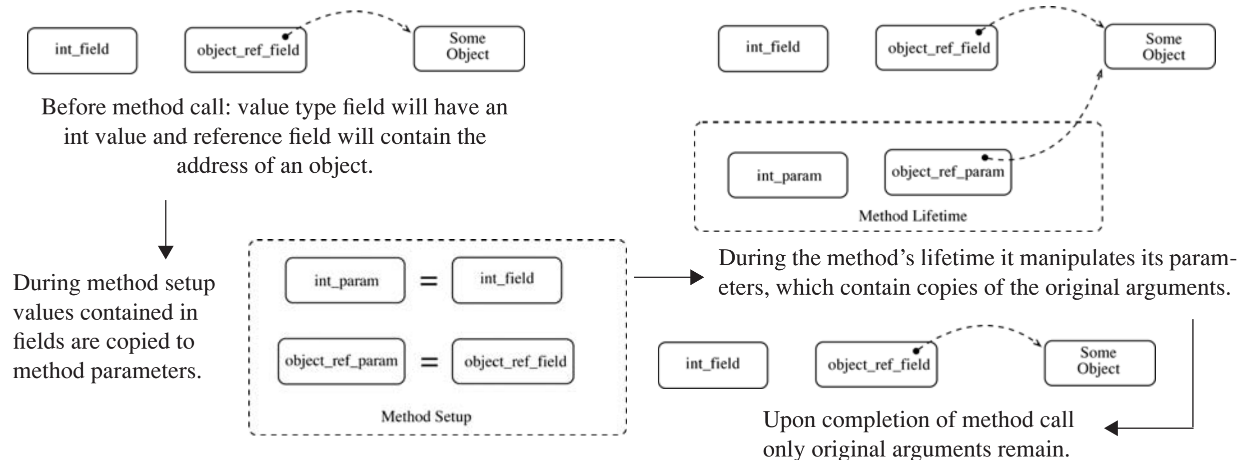


Figure 9-17: Default Value Parameter Behavior

Referring to Figure 9-17 — Prior to a method call, value type and reference fields contain values. During method setup 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, value types and references used as arguments will retain their original values. Changes to the object pointed to by the reference parameter will remain in effect.

Reference Parameters: Using The `ref` Parameter Modifier

The `ref` modifier can be applied to parameters to change the way they behave inside of a method. Consider for a moment this modified version of SomeMethod():

```
public void SomeMethod(ref int int_param, ref Object object_ref_param){
      // body statements omitted
}
```

In this version, the `ref` modifier is applied to each parameter. Figure 9-18 illustrates how these reference parameters behave differently from value parameters.

Referring to Figure 9-18 — the `ref` modifier changes the behavior of the method's parameters. Value type arguments like int, float, double, etc., behave as though they are references. Any change made to a value type ref parameter in the body of the method affects the original argument. In the case of reference type arguments, the `ref` parameter is a reference to a reference, as you can see from the diagram. What this means is that if you create a new object in the body of the method and assign its address to a reference parameter, it will be assigned to the original argument reference and it will now point to the new object.

I see that glazed look in your eyes. Check out the following two programs and note their behavior. Example 9.24 demonstrates the scenario shown in Figure 9-17. Example 9.25 demonstrates the scenario shown in Figure 9-18.

© 2008 Rick Miller — All Rights Reserved

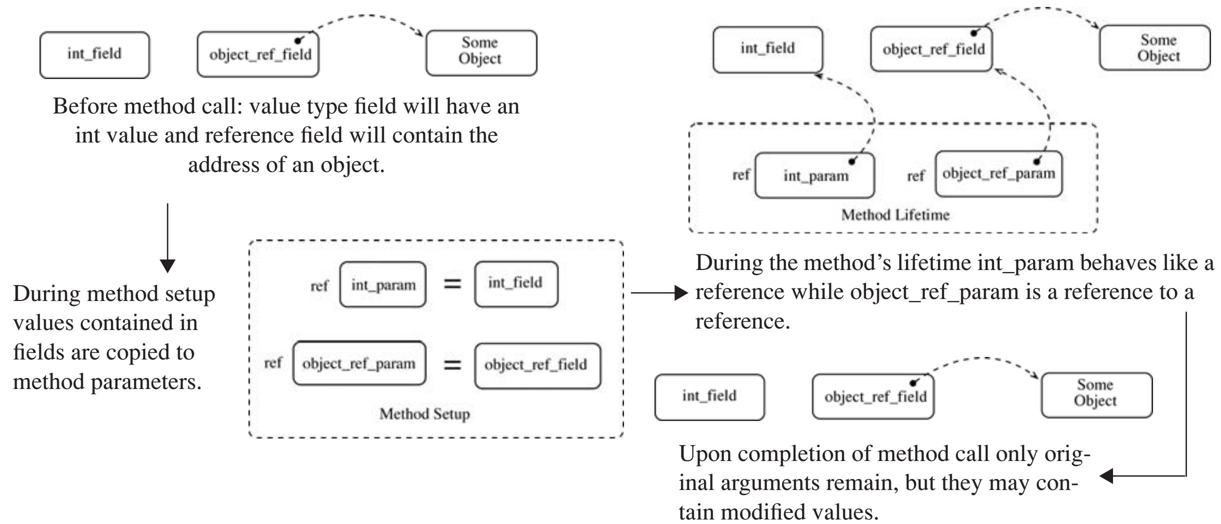


Figure 9-18: Reference Parameter Behavior — Using `ref` Modifier

9.24 ValueParameterTest.cs

```
1    using System;
2    using System.Text;
3
4    public class ValueParameterTest {
5
6      int int_field;
7      StringBuilder object_ref_field = new StringBuilder();
8
9      public void F(int int_param, StringBuilder object_ref_param){
10        int_param = 2;
11        Console.WriteLine("Value of int_param modified in method: " + int_param);
12        object_ref_param.Append("Two");
13        Console.WriteLine("The value of object_ref_param after calling Append() in method: "
14                          + object_ref_param);
15        object_ref_param = new StringBuilder();
16        object_ref_param.Append("Three");
17        Console.WriteLine("The value of object_ref_param after calling Append() in method: "
18                          + object_ref_param);
19
20     }
21
22     public void G(){
23        int_field = 1;
24        object_ref_field.Append("One");
25        Console.WriteLine("The value of int_field before method call is: " + int_field);
26        Console.WriteLine("The value of the object_ref_field before method call is: " + object_ref_field);
27        Console.WriteLine("--------------------------------------------------------------------------");
28        F(int_field, object_ref_field);
29        Console.WriteLine("--------------------------------------------------------------------------");
30        Console.WriteLine("The value of int_field after method call is: " + int_field);
31        Console.WriteLine("The value of the object_ref_field after method call is: " + object_ref_field);
32
33     }
34
35     public static void Main(){
36        ValueParameterTest pt = new ValueParameterTest();
37        pt.G();
38     } // end Main
39   } // end class definition
```

Referring to Example 9.24 — the ValueParameterTest class declares two fields: int_field and object_ref_field. The object_ref_field is of type StringBuilder. The method F() whose definition begins on line 9 declares two parameters, one of type int named int_param and one of type StringBuilder named object_ref_param. The important thing to note in the body of method F() is that after the Append() method is called on the initial object_ref_param value, a new StringBuilder is created on line 15 and its reference is assigned to object_ref_param. Note that this has no effect on the reference value contained in object_ref_field.

Method G() whose definition begins on line 22 simply prints field values to the console before and after calling method F(). Again, value parameter passing is the default mode. Compare this code with Example 9.25.

 © 2008 Rick Miller — All Rights Reserved

9.25 RefParameterTest.cs

```
1    using System;
2    using System.Text;
3
4    public class RefParameterTest {
5
6      int int_field;
7      StringBuilder object_ref_field = new StringBuilder();
8
9      public void F(ref int int_param, ref StringBuilder object_ref_param){
10        int_param = 2;
11        Console.WriteLine("Value of int_param modified in method: " + int_param);
12        object_ref_param.Append("Two");
13        Console.WriteLine("The value of object_ref_param after calling Append() in method: "
14                           + object_ref_param);
15        object_ref_param = new StringBuilder();
16        object_ref_param.Append("Three");
17        Console.WriteLine("The value of object_ref_param after calling Append() in method: "
18                           + object_ref_param);
19
20     }
21
22     public void G(){
23        int_field = 1;
24        object_ref_field.Append("One");
25        Console.WriteLine("The value of int_field before method call is: " + int_field);
26        Console.WriteLine("The value of the object_ref_field before method call is: " + object_ref_field);
27        Console.WriteLine("--------------------------------------------------------------------------");
28        F(ref int_field, ref object_ref_field);
29        Console.WriteLine("--------------------------------------------------------------------------");
30        Console.WriteLine("The value of int_field after method call is: " + int_field);
31        Console.WriteLine("The value of the object_ref_field after method call is: " + object_ref_field);
32     }
33
34     public static void Main(){
35        RefParameterTest pt = new RefParameterTest();
36        pt.G();
37     } // end Main
38   } // end class definition
```

Referring to Example 9.25 — the differences between this code and the previous example is the following: 1) the class name, 2) the `ref` modifier has been applied to both of method F()'s parameters, and 3) the `ref` argument modifier has been applied to the arguments passed to the F() method call on line 28. This is required otherwise you will receive a compiler error. Figures 9.19 and 9.20 show the results of running these programs.

```
Command Prompt

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ParameterBehaviorTest>valueparametertest
The value of int_field before method call is: 1
The value of the object_ref_field before method call is: One
--------------------------------------------------------------------------
Value of int_param modified in method: 2
The value of object_ref_param after calling Append() in method: OneTwo
The value of object_ref_param after calling Append() in method: Three
--------------------------------------------------------------------------
The value of int_field after method call is: 1
The value of the object_ref_field after method call is: OneTwo

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ParameterBehaviorTest>_
```

Figure 9-19: Results of Running Example 9.24

```
Command Prompt

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ParameterBehaviorTest>refparametertest
The value of int_field before method call is: 1
The value of the object_ref_field before method call is: One
--------------------------------------------------------------------------
Value of int_param modified in method: 2
The value of object_ref_param after calling Append() in method: OneTwo
The value of object_ref_param after calling Append() in method: Three
--------------------------------------------------------------------------
The value of int_field after method call is: 2
The value of the object_ref_field after method call is: Three

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ParameterBehaviorTest>_
```

Figure 9-20: Results of Running Example 9.25

© 2008 Rick Miller — All Rights Reserved

The `out` Parameter Modifier

The `out` parameter modifier indicates that a parameter will be used to return a result to the calling program via its associated argument. An `out` parameter is similar to a `ref` parameter, but differs in that the initial value of an out parameter's associated argument is not important. Example 9.26 shows the use of the out parameter modifier.

9.26 OutParamTest.cs

```
1    using System;
2
3    public class OutParamTest {
4      int  _a = 2;
5      int  _count = 10;
6      long _result;
7
8      public void Factor(int value, int power, out long total){
9        total = 1;
10       for(int i = 1; i <= power; i++){
11         total = total * value;
12         Console.WriteLine("Value of i is {0} and value of total is {1}", i, total);
13       }
14     }
15
16     public void Run(){
17       Console.WriteLine("The value of _result before calling Factor is: " + _result);
18       Console.WriteLine("-----------------------------------------------------------");
19       Factor(_a, _count, out _result);
20        Console.WriteLine("-----------------------------------------------------------");
21       Console.WriteLine("The value of _result after calling Factor is: " + _result);
22     }
23
24     public static void Main(){
25       OutParamTest pt = new OutParamTest();
26       pt.Run();
27     } // end Main
28   } // end class definition
```

Referring to Example 9.26 — the class defines three fields: _a, _count, and _result. The Factor() method whose definition begins on line 8 declares three parameters, the last of which is an out parameter named total. The Run() method on line 16 writes the value of _result to the console before and after the Factor() method call. Figure 9-21 shows the results of running this program.

```
Command Prompt
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\OutParamTest>outparamtest
The value of _result before calling Factor is: 0
-----------------------------------------------------------
Value of i is 1 and value of total is 2
Value of i is 2 and value of total is 4
Value of i is 3 and value of total is 8
Value of i is 4 and value of total is 16
Value of i is 5 and value of total is 32
Value of i is 6 and value of total is 64
Value of i is 7 and value of total is 128
Value of i is 8 and value of total is 256
Value of i is 9 and value of total is 512
Value of i is 10 and value of total is 1024
-----------------------------------------------------------
The value of _result after calling Factor is: 1024

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\OutParamTest>
```

Figure 9-21: Results of Running Example 9.26

Parameter Arrays: Using The `params` Modifier

Methods can take an indefinite number of arguments with the help of parameter arrays. Use the `params` modifier to declare an array parameter argument. If a parameter array appears in a method's parameter list, it must be either the last parameter in the list or the only parameter. Example 9.27 offers a short program that demonstrates the use of a parameter array.

9.27 ParamArrayTest.cs

```
1    using System;
2
3    public class ParamArrayTest {
4
5      public void ParamMethod(params String[] args){
6         Console.WriteLine("Method called with {0} arguments.", args.Length);
7         for(int i = 0; i<args.Length; i++){
```


© 2008 Rick Miller — All Rights Reserved

```
8           Console.WriteLine("Argument " + i + " is " + args[i]);
9         }
10      }
11
12      public static void Main(){
13        ParamArrayTest pt = new ParamArrayTest();
14        pt.ParamMethod();
15        pt.ParamMethod("one");
16        pt.ParamMethod("one", "two");
17        pt.ParamMethod(new String[] {"one", "two", "three"});
18      }
19    }
```

Referring to Example 9.27 — the ParamMethod() whose definition begins on line 5 declares a String parameter array. The method simply prints the number of arguments it was called with, and then prints the value of each argument to the console. The method's use is demonstrated in the Main() method. The ParamMethod() is called on line 14 with no arguments, followed by a call with one argument, next with two arguments, then finally with a String array that contains three arguments. I included this last method call to show you how arguments can also be passed as an array of the type expected by the method. Figure 9-22 shows the results of running this program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ParamArrays>paramarraytest
Method called with 0 arguments.
Method called with 1 arguments.
Argument 0 is one
Method called with 2 arguments.
Argument 0 is one
Argument 1 is two
Method called with 3 arguments.
Argument 0 is one
Argument 1 is two
Argument 2 is three

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ParamArrays>
```

Figure 9-22: Results of Running Example 9.27

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, better still, 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 of 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 value or reference type you require. For reference types, the `new` keyword can be used to create argument objects on the fly. 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 an 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());
```


© 2008 Rick Miller — All Rights Reserved

As you progress through this book and your knowledge of C# grows, you will be exposed to all the above forms of a method call plus several more.

Quick Review

By default, 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. Any change to the parameter values only affect the copies, not the actual arguments. Changes to an object pointed to by a reference parameter will affect the original object. However, a change to what a reference parameter points to only affects the parameter, not the original reference argument.

Use the `ref` parameter modifier to change parameter behavior so that changes made to the parameters also affect the original argument values. By using the `ref` parameter, a change to what a reference parameter points to (*i.e.,* creating a new object with the new operator) will also change what the original reference argument points to.

Use the `out` parameter modifier if you need to return method results via one or more of its arguments.

Use the `params` modifier to create parameter arrays.

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.

Structures vs. Classes

Structures (structs) share many similarities with classes with one huge difference; a structure defines a new value type whereas a class defines a new reference type, as is shown in Figure 9-23. This section highlights the differences between structures and classes and offers some advice on when you might want to use a structure vs. a class.

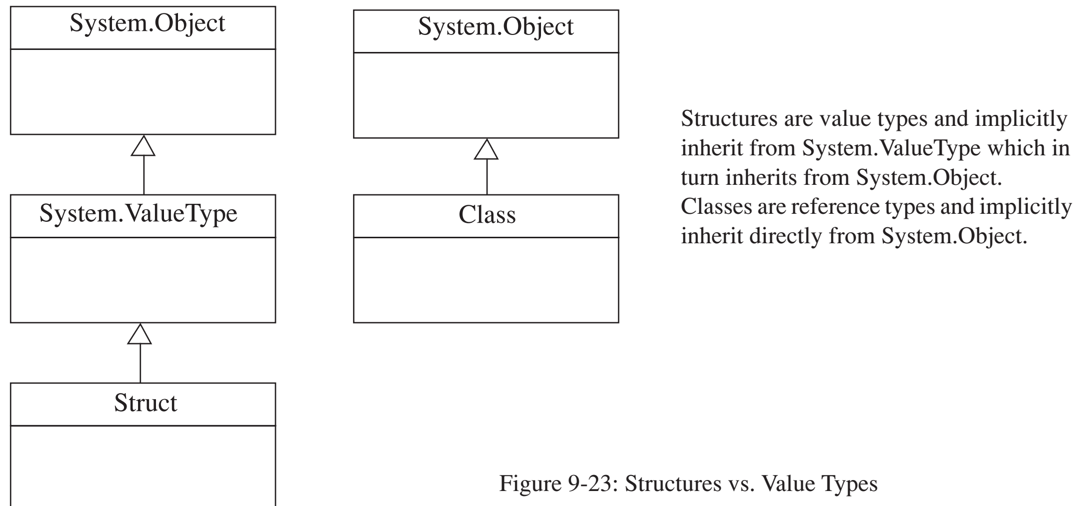


Figure 9-23: Structures vs. Value Types

Value Semantics vs. Reference Semantics

A structure type (value type) variable directly contains the data associated with the structure as opposed to a class type (reference type) variable that contains a reference to an object in memory. It is possible for two different reference variables to point to the same object in memory. But not so for value type variables, where each variable has its own copy of the data.

Structures are not allocated on the heap unless they undergo a *boxing* operation. Boxing and unboxing are covered below.

Because structure variables are not reference variables they cannot be null.

© 2008 Rick Miller — All Rights Reserved

Ten Authorized Members vs. Eleven

Structures can have *constants*, *fields*, *methods*, *properties*, *events*, *indexers*, *operators*, *constructors*, *static constructors*, and *nested type declarations*.

A structure cannot have a finalizer, nor can you define an explicit parameterless (default) constructor.

Default Variable Field Values

As stated above, you cannot define a parameterless (default) constructor for a structure. The compiler-supplied default constructor will set all a structure's value type fields to their default values and any reference type fields to null. Also, you cannot use instance field initializers to set the values of each field.

Behavior During Assignment

The assignment of one value type variable to another causes a *complete copy of the structure's data* being assigned. Compare this behavior to that of a reference type where only the *reference to an object* is copied from the variable being assigned. Recall that when value types are passed as arguments to methods, a copy of the argument is assigned to its corresponding parameter. This behavior was discussed in detail earlier in this chapter. (*Also, see Chapter 22 — Well-Behaved Objects*)

`this` Behaves Differently

In a structure, `this` is considered a variable via which the values of the structure can be assigned to and modified. In an instance constructor, `this` functions like an `out` parameter. In an instance method, `this` functions like a `ref` parameter.

Inheritance Not Allowed

You cannot extend a structure. Structures are never abstract and always inherently sealed. Structures can implement interfaces but they cannot specify a base class. Structure members cannot be abstract or virtual. The override keyword is only allowed when overriding members of System.ValueType.

Boxing And Unboxing

If you need to treat a value type like a reference type, you can *box* the value-type into an object that is then allocated on the heap. Look at the following example.

9.28 BoxingDemo.cs

```
1    using System;
2
3    public class BoxingDemo {
4
5      public static void Main(){
6        int i = 3;
7        Console.WriteLine("Unboxed i =  " + i);
8        object o = i; // boxing
9        Console.WriteLine("o = " + o);
10       o = 4;       // treat o like an int
11       Console.WriteLine("Modified o = " + o);
12       Console.WriteLine("Unboxed i = " + i);
13       i = (int)o; // unboxing
14       Console.WriteLine("Modified i = " + i);
15     }
16   }
```

Referring to Example 9.28 — the local variable i is declared and initialized to the value 3. On line 8, an object reference named o is declared and the value-type i is boxed by the assignment to o. The reference o now points to a boxed integer value type. Line 10 demonstrates that assignments to o can take place like assignments to ordinary integers. On line 13, the value type contained in o is unboxed and assigned to the variable i. The explicit cast is required. Figure 9-24 shows the results of running this program.

© 2008 Rick Miller — All Rights Reserved

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\BoxingDemo>boxingdemo
Unboxed i =  3
o = 3
Modified o = 4
Unboxed i = 3
Modified i = 4

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\BoxingDemo>
```

Figure 9-24: Results of Running Example 9.28

When To Use Structures

Structures are appropriate when the amount of data they contain is small. Remember, when one structure variable is assigned to another a complete copy of the assigned structure's data is copied over. The ultimate answer to the structure vs. class question can only be answered by thoroughly accessing a project's design and performance requirements.

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 can be implemented as *structures* or *classes*. These structures or classes will interact with each other in some capacity to implement the complete problem solution.

A UML class diagram shows the static relationship between classes that participate in a software design. Programmers use the class diagram to express and clarify design concepts to themselves, to other programmers, to management, and to clients.

In UML, a rectangle represents a class. 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.

A stereotype introduces a new type of element within a system. The stereotype name is contained within the guillemet characters << >>.

Generalization and *specialization* are indicated by 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.

C# classes can contain eleven different types of members: *fields, constants*, *methods*, *properties*, *events*, *indexers*, *operators*, *instance constructors*, *static constructors, finalizers*, and *nested type declarations*.

The access modifiers `public`, `protected`, `private`, `internal`, and `protected internal` are used to control access to class and instance members. If no access is specified then private is assumed.

The term *horizontal access* describes the access a client object has to the members 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 members via the class name, or 2) by creating an instance of the class and accessing its public non-static members via an object reference.

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.

© 2008 Rick Miller — All Rights Reserved

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 set up or build an object when it's created in memory. If you do not provide one, the compiler will create a default constructor for you, but it may or may not provide the level of functionality you require.

Incrementally build and test abstract data types by iteratively applying the steps of the development cycle. Start with the class definition shell and then add fields, properties, and methods as required to fulfill the class's design objectives.

Test class functionality with the help of a *test driver*. A test driver is a small program that's used to exercise the functionality of another program.

By default, 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. Any change to the parameter values only affect the copies, not the actual arguments. Changes to an object pointed to by a reference parameter will affect the original object. However, a change to what a reference parameter points to only affects the parameter, not the original reference argument.

Use the `ref` parameter modifier to change parameter behavior so that changes made to the parameters also affect the original argument values. By using the `ref` parameter, a change to what a reference parameter points to (*i.e.,* creating a new object with the `new` operator) will also change what the original reference argument points to.

Use the `out` parameter modifier if you need to return method results via one or more of its arguments.

Use the `params` modifier to create parameter arrays.

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.

Skill-Building Exercises

1. **.NET API Drill:** Browse through the .NET 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.

© 2008 Rick Miller — All Rights Reserved

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 value type and reference fields. It should also contain several methods that return value types and references. Write a method that takes at least one value 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. Change the method's parameter behavior with the use of the `ref` and `out` modifiers. Again, write some code that calls the method and note the results on both the arguments supplied to the method and to the method's parameters during the method's lifetime.

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 gasoline 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 reference 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 Gasoline 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 gasoline 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

© 2008 Rick Miller — All Rights Reserved

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 International Standard Book Number (ISBN). You can add any other attributes you deem necessary. Create a class named LibraryManager that lets you create and add books to your library, delete books from your library, and list the books 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 C# 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:

9.29 Node.cs (Partial Listing)

```
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    }
```

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 creates data structures designed for use within a linked list. Use the code shown in Example 9.29 to help you write a program that manages a linked list. Here are a few hints to get you started:

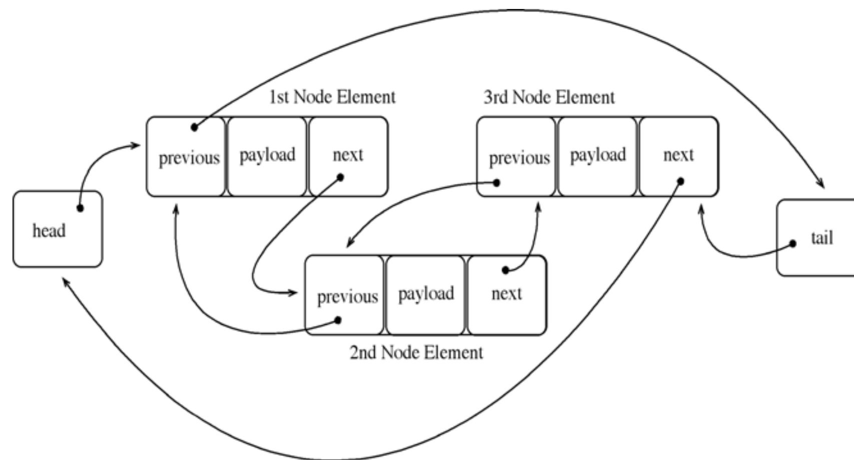


Figure 9-25: Circular Linked List with Three Nodes

Referring to Figure 9-25 above — a linked list contains one or more 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 9-25 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, it can be used to create circular linked list as is shown in Figure 9-25.

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 or properties to the Node class code given in Example 9.29. At a minimum you should add properties 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.

 © 2008 Rick Miller — All Rights Reserved

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. What are the eleven categories of C# class members?

7. What's the difference between static and non-static fields?

8. What the difference between static and non-static methods?

9. What's the difference between `readonly` fields and `const` fields?

10. List and describe the purpose of member access modifiers.

11. Explain the concept of horizontal access. Draw a picture showing how client code access to a server class's members is controlled using the access modifiers `public` and `private`.

12. What is a method?

13. List and describe the desirable characteristics of a method.

14. Explain the concept of *cohesion* as it pertains to methods.

15. (True/False) A method should be maximally cohesive.

16. What steps can you take as a programmer to ensure your methods are maximally cohesive?

17. What's the purpose of a method definition?

18. What parts of a method definition are optional?

19. What is meant by the term method signature?

20. What parts of a method are included in a method's signature?

21. What constitutes an overloaded method?

22. Give at least one example for which method overloading is useful.

© 2008 Rick Miller — All Rights Reserved

23. What makes constructor methods different from ordinary methods?

24. Describe in your own words how arguments are passed to methods.

References

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Rick Miller. *Java For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-05-2

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

© 2008 Rick Miller — All Rights Reserved

Chapter 10

Pentax 67 / SMC Takumar 90/2.8 / Kodak Tri-X Professional

Flowers

Compositional Design

Learning Objectives

- State the definition of the term "simple aggregation"
- State the definition of the term "composite aggregation"
- Explain the difference between simple and composite aggregation
- Express simple and composite aggregation using Unified Modeling Language (UML) diagrams
- Express an association between two class types using a UML diagram
- Define the terms "has a", "contains", and "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 both 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

 © 2008 Rick Miller — All Rights Reserved

Introduction

Rarely does an application comprise just one class. In reality, applications are typically 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 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 UML.

The physical complexity of the programs you write in this chapter must be managed through source file organization. If you have not already 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 an integrated development environment (IDE) like Microsoft's Visual Studio, it will manage the source file dependencies automatically. If you are developing your projects with a simple text editor and the .NET Runtime Environment as recommended in Chapter 2, then you can use Microsoft Build (MSBuild.exe) located in the .NET Framework folder. MSBuild is what Visual Studio uses to build projects. However, you don't need to use these tools just yet. As the next section explains, you can compile almost all of the projects in this book from the command line with the `csc` compiler tool.

Compiling Multiple Source Files Simultaneously With csc

If you don't have time to learn MSBuild and don't want to bother with Visual Studio just yet, that's no problem. You can compile multiple source files simultaneously using the `csc` compiler tool. All you need to do is enter the names of the source files you want to compile following the `csc` command on the command line, as is shown in the following example:

```
csc SourceFileOne.cs SourceFileTwo.cs SourceFileThree.cs
```

List each source file 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.

© 2008 Rick Miller — All Rights Reserved

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 (Microsoft Windows) or a shell script (Linux, Mac 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 and invoke the `csc` compiler tool as shown in the following example:

```
csc *.cs
```

This compiles all the source files in a given directory. You may, of course, provide an explicit directory path as shown in the following example:

```
csc c:\myprojects\project1\*.cs
```

If you have organized your project files into multiple subdirectories, you can use the `/recurse` compiler switch to recursively compile the source files. Recursive compilation starts with files located in the root project directory, and visits each subdirectory in turn. To compile recursively, use the `/recurse` compiler switch like so:

```
csc /recurse:*.cs
```

Quick Review

There are two types of complexity: *conceptual* and *physical*. Manage conceptual complexity by using object-oriented design concepts and by expressing your object-oriented designs in UML. Manage physical complexity with the help of your IDE or with an automated build tool such as Microsoft Build (MSBuild). 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 `csc` compiler tool.

Dependency vs. Association

A C# program built from many classes manifests 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 effect on the behavior of the class or classes that depend on it. For example, say you have two classes, Class A and Class B. If Class A depends upon the behavior of Class B in a way that a change to Class B affects 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 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 .NET 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 them, 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 comprising other objects (*i.e.,* the whole object) is referred to as an *aggregate*. Objects used to build the whole object are

 © 2008 Rick Miller — All Rights Reserved

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 are 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 is how the whole or aggregate object is linked to its part objects. The type of linking between an aggregate and its parts dictates 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. This section discusses simple and composite aggregation in greater detail.

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) independently 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 .NET runtime garbage collector eventually destroys unreferenced objects, but as 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 C#, there is no direct way for a programmer to destroy an object. You must rely on the .NET runtime 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.

© 2008 Rick Miller — All Rights Reserved

Simple Aggregation Expressed In UML

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 diamond 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.

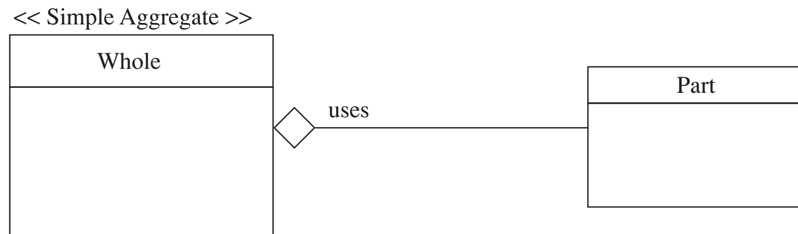


Figure 10-1: UML Diagram Showing Simple Aggregation

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.

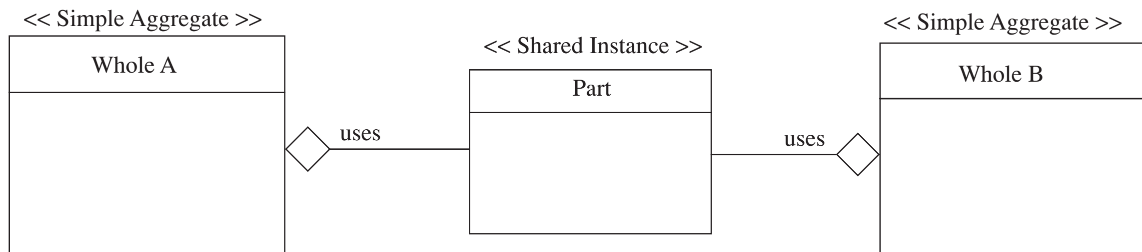


Figure 10-2: Part Class Shared Between Simple Aggregate Classes

Composite Aggregation Expressed In UML

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.

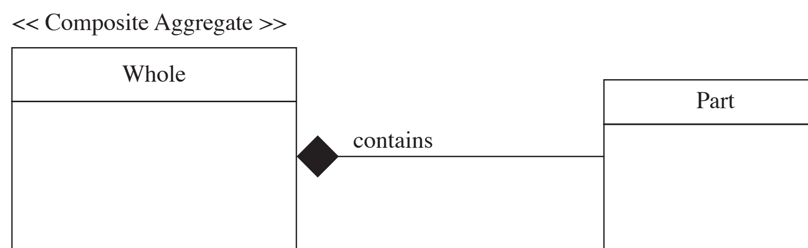


Figure 10-3: UML Diagram Showing Composite Aggregation

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.

 © 2008 Rick Miller — All Rights Reserved

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.

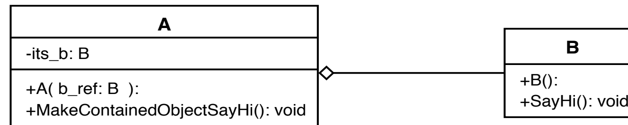


Figure 10-4: Simple Aggregation Example

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() that returns `void`.

Class B has no attributes, one constructor method, and another method named SayHi(). Examples 10.1 and 10.2 give the code for these two classes.

10.1 A.cs

```
1    using System;
2
3    public class A {
4      private B its_b = null;
5
6      public A(B b_ref){
7        its_b = b_ref;
8        Console.WriteLine("A object created!");
9      }
10
11     public void MakeContainedObjectSayHi(){
12       its_b.SayHi();
13     }
14   }
```

Referring to Example 10.1 — the its_b reference variable is declared on line 4 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 12 in the MakeContainedObjectSayHi() method to call its SayHi() method.

10.2 B.cs

```
1    using System;
2
3    public class B {
4      public B(){
5        Console.WriteLine("B object created!");
6      }
7
8      public void SayHi(){
9        Console.WriteLine("Hi!");
10     }
11   }
```

The only thing to note in Example 10.2 is the SayHi() method, which starts on line 8. 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.cs

```
1    public class TestDriver {
2      public static void Main(){
3        B b = new B();
4        A a = new A(b);
5        a.MakeContainedObjectSayHi();
6      }
7    }
```

 © 2008 Rick Miller — All Rights Reserved

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 a call is made to the MakeContainedObjectSayHi() method via reference a. The results of running this program are shown in Figure 10-5.

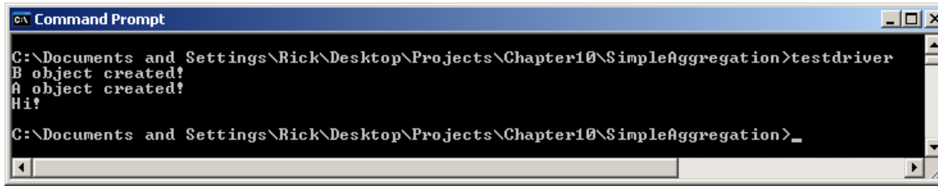


Figure 10-5: Results of Running Example 10.3

Composite Aggregation Example

Figure 10-6 gives a UML diagram that illustrates composite aggregation between classes A and B.

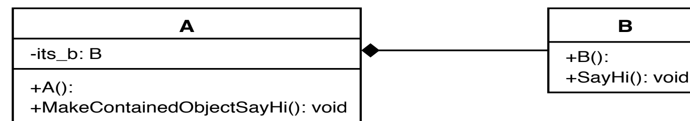


Figure 10-6: Composite Aggregation Example

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.

10.4 A.cs

```
1     using System;
2
3     public class A {
4       private B its_b = null;
5
6       public A(){
7         its_b = new B();
8         Console.WriteLine("A object created!");
9       }
10
11      public void MakeContainedObjectSayHi(){
12        its_b.SayHi();
13      }
14    }
```

10.5 B.cs

```
1     using System;
2
3     public class B {
4       public B(){
5         Console.WriteLine("B object created!");
6       }
7
8       public void SayHi(){
9         Console.WriteLine("Hi!");
10      }
11    }
```

Referring to Example 10.4 — the its_b attribute is declared and initialized to null on line 4. In the A() constructor on line 7, 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.

© 2008 Rick Miller — All Rights Reserved

Example 10.6 gives a modified version of the TestDriver program. Compare Example 10.6 to Example 10.3. 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.

10.6 TestDriver.cs

```
1    public class TestDriver {
2      public static void Main(){
3        A a = new A();
4        a.MakeContainedObjectSayHi();
5      }
6    }
```

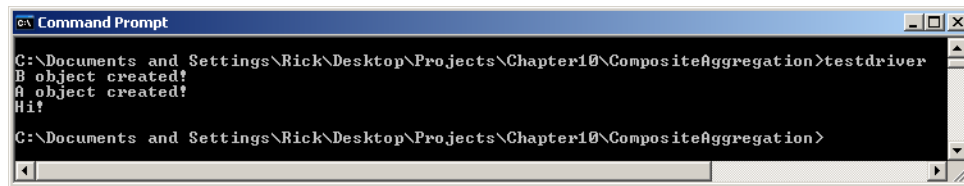


Figure 10-7: Results of Running Example 10.6

Quick Review

The UML class diagram can be used to show aggregation associations between classes. A hollow diamond denotes simple aggregation and expresses a *uses* or *uses a* relationship between the whole and part classes. A solid diamond denotes composite aggregation and expresses a *contains* or *has a* relationship between whole and part classes.

Sequence Diagrams

A sequence diagram is another type of UML diagram that illustrates 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 program given in Examples 10.1 through 10.3 in the previous section.

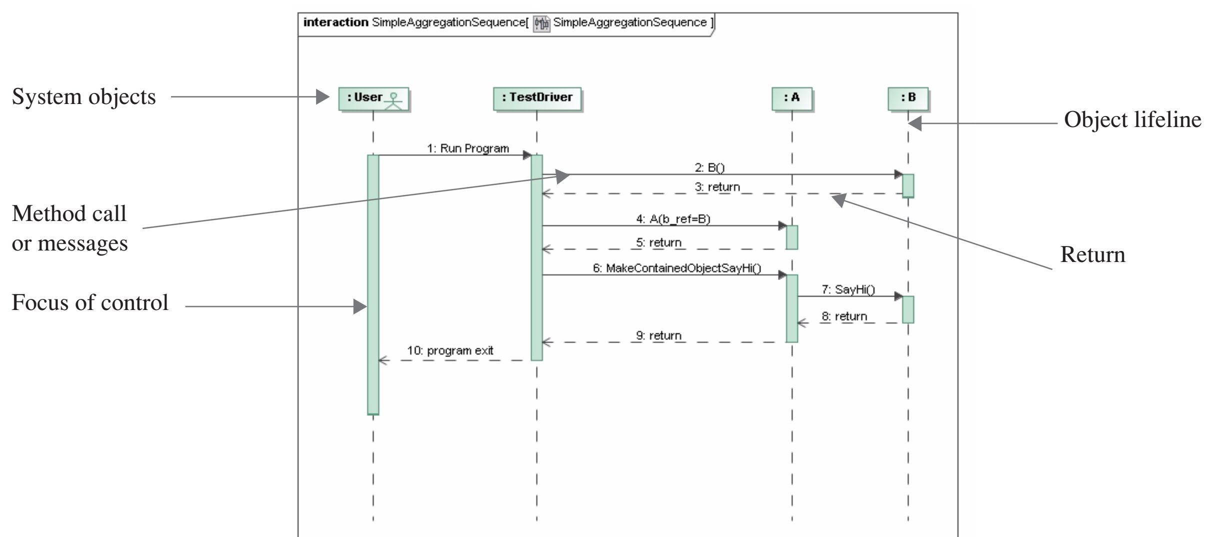


Figure 10-8: Sequence Diagram — Simple Aggregation

 © 2008 Rick Miller — All Rights Reserved

Referring to Figure 10-8 — read the sequence diagram 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. 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.,* It calls a method.) 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.

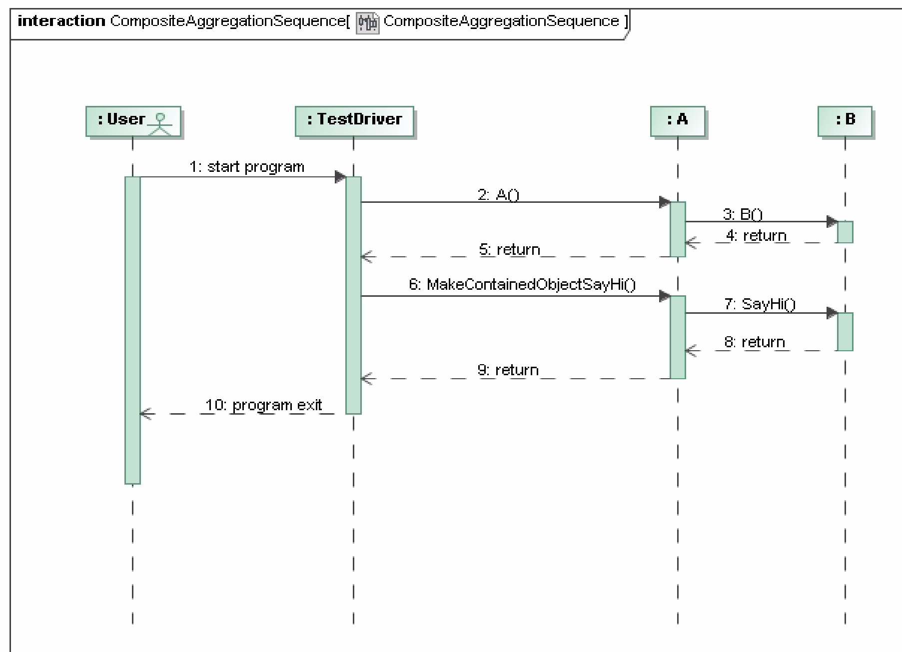


Figure 10-9: Sequence Diagram — Composite Aggregation

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.

The sequence diagrams shown here are unique in that the example program is small enough to show the whole sequence of events in one picture. In reality, however, most software systems are so complex that sequence diagrams usually focus on one piece of functionality at a time. You'll see an example of this in the next section.

Magic Draw

I used a UML design tool called Magic Draw to create the sequence diagrams in this chapter. You can get more information about Magic Draw at [www.magicdraw.com]

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.

© 2008 Rick Miller — All Rights Reserved

The Engine Simulation: An Extended Example

In this section I will ramp up the complexity somewhat and present an extended example of compositional design: the 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 engine simulation. Read it carefully before proceeding to the next section.

```
                    Project Specification
                      Engine Simulation

Objectives:
- Apply compositional design techniques to create a C# 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


Tasks:
- Write a program that simulates the basic functionality of an 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, the engine
  should stop running if it detects a fault in one of its parts.
- When an individual part's status changes, the engine must perform a
  comprehensive status check on itself.
- If, when trying to start, 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: Engine Simulation Project Specification

The project specification offers good direction and several hints regarding the project requirements. The 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, OilPump, Compressor, OxygenSensor, and TemperatureSensor classes respectively. Each of these classes has an association with the PartStatus enumeration, as the class diagram in Figure 10-11 illustrates.

 © 2008 Rick Miller — All Rights Reserved

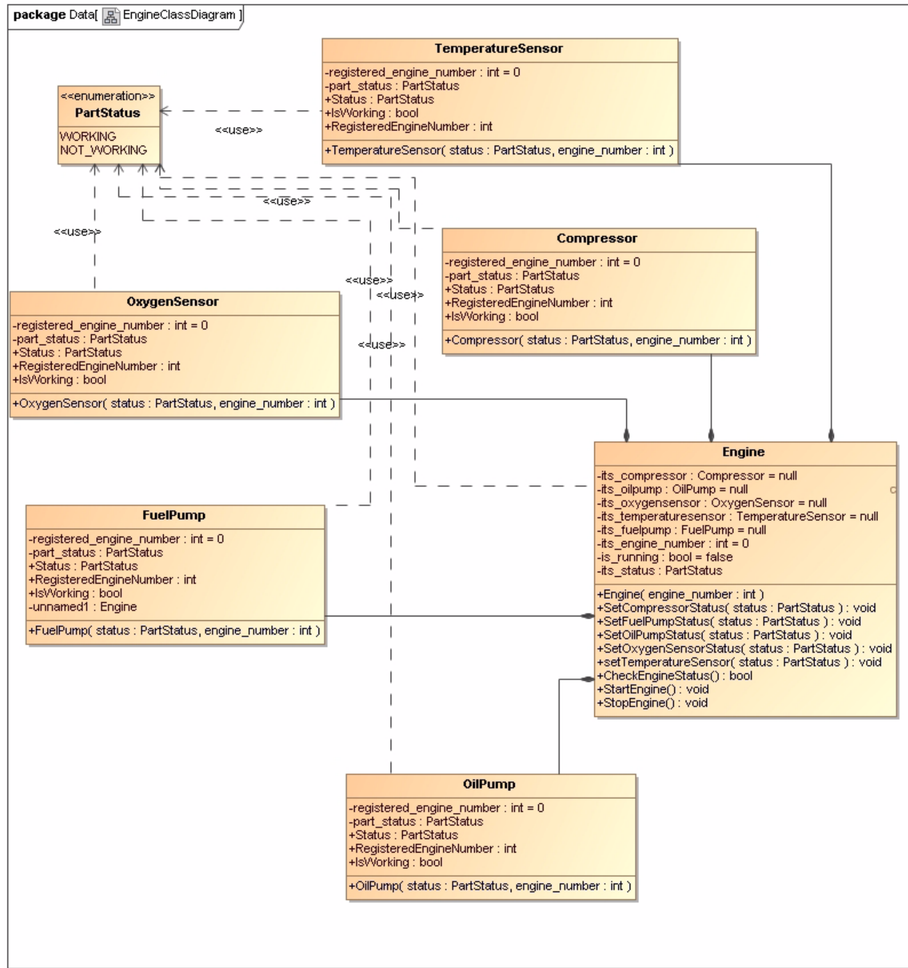


Figure 10-11: Engine Simulation Class Diagram

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.

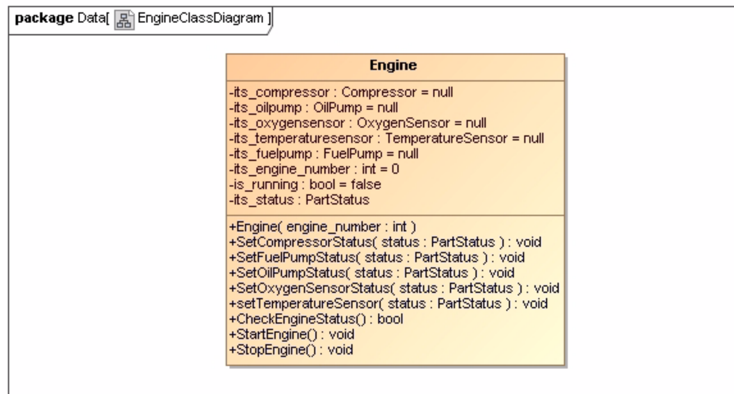


Figure 10-12: Engine Class Diagram

© 2008 Rick Miller — All Rights Reserved

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 maps to its respective part class. However, several more attributes are required to complete the class. Two of these, *its_engine_number* and *is_running,* are value type variables. The last attribute, *its_status*, is an enumeration variable of type PartStatus.

As you can tell from looking at the class diagram in Figure 10-12, all of 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 guards against returning a reference to any of its part objects. This ensures 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 garbage collector. Although, if you remember, you cannot guarantee when this garbage 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.

Engine Simulation Sequence Diagrams

The 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 simulation program. In fact, this diagram only accounts for the sequence of events resulting from the execution of line 3 of Example 10.7 in the next section. The creation of additional sequence diagrams for the simulation program is left for you to do as an exercise.

Running The Engine Simulation Program

To run the 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 10-7. The result of running this program is shown in Figure 10-14.

10.7 EngineTester.cs

```
1    using EngineSimulation;
2
3    public class EngineTester {
4      public static void Main(){
5       Engine e1 = new Engine(1);
6       e1.StartEngine();
7       e1.SetCompressorStatus(PartStatus.NOT_WORKING);
8       e1.StartEngine();
9       e1.SetCompressorStatus(PartStatus.WORKING);
10      e1.StartEngine();
11     }
12   }
```

Referring to Example 10.7 — an Engine object is created on line 5. As noted above, this line kicks off the sequence of events illustrated in Figure 10-13. All of the Engine's part objects are initially created with WORKING part status. The StartEngine() method call on line 6 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 7, a fault is introduced to the engine's compressor component. This causes the engine to shut down. The attempt on line 8 to start the engine fails 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.

© 2008 Rick Miller — All Rights Reserved

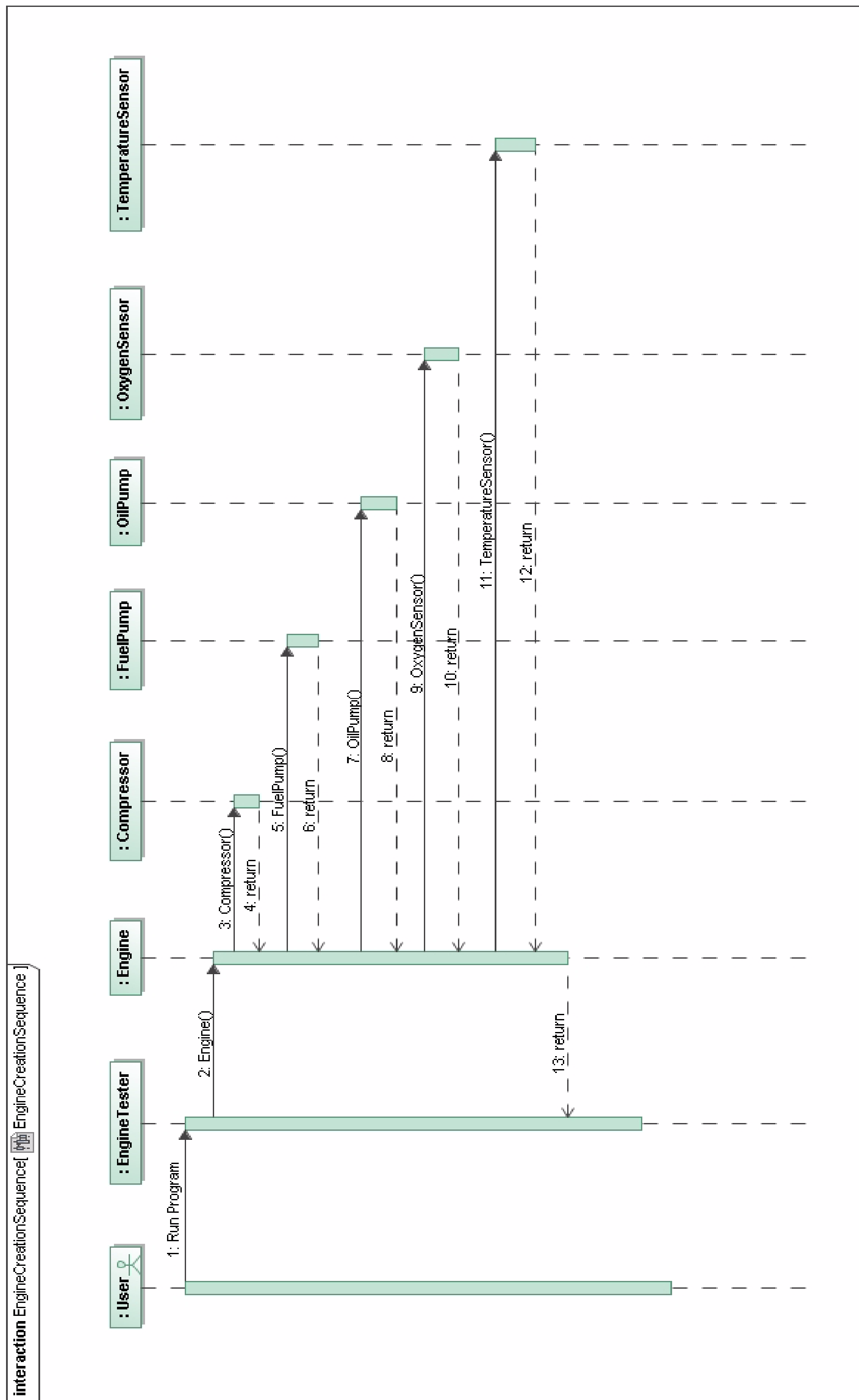


Figure 10-13: Create Engine Object Sequence

 © 2008 Rick Miller — All Rights Reserved

```
Command Prompt
C:\Documents and Settings\Rick\Desktop\Projects\Chapter10\Engine>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.
All engine #1 components working properly.
All engine #1 components working properly.
Engine #1 is running!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter10\Engine>
```

Figure 10-14: Result of Running Example 10.7

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 simulation use the functionality of the PartStatus enumeration.

Complete Engine Simulation Code Listing

10.8 Engine.cs

```
1    using System;
2
3    namespace EngineSimulation {
4
5        public class Engine {
6            private Compressor its_compressor = null;
7            private FuelPump its_fuelpump = null;
8            private OilPump its_oilpump = null;
9            private OxygenSensor its_oxygensensor = null;
10           private TemperatureSensor its_temperaturesensor = null;
11           private int its_engine_number = 0;
12           private bool is_running = false;
13           private PartStatus its_status;
14
15           public Engine(int engine_number) {
16               its_engine_number = engine_number;
17               its_compressor = new Compressor(PartStatus.WORKING, its_engine_number);
18               its_fuelpump = new FuelPump(PartStatus.WORKING, its_engine_number);
19               its_oilpump = new OilPump(PartStatus.WORKING, its_engine_number);
20               its_oxygensensor = new OxygenSensor(PartStatus.WORKING, its_engine_number);
21               its_temperaturesensor = new TemperatureSensor(PartStatus.WORKING, its_engine_number);
22               its_status = PartStatus.WORKING;
23               Console.WriteLine("Engine #" + its_engine_number + " created!");
24           }
25
26           public void SetCompressorStatus(PartStatus status) {
27               its_compressor.Status = status;
28               CheckEngineStatus();
29           }
30
31           public void SetFuelPumpStatus(PartStatus status) {
32               its_fuelpump.Status = status;
33               CheckEngineStatus();
34
35           }
36
37           public void SetOilPumpStatus(PartStatus status) {
38               its_oilpump.Status = status;
39               CheckEngineStatus();
40           }
41
```

 © 2008 Rick Miller — All Rights Reserved

```
42          public void SetOxygenSensorStatus(PartStatus status) {
43              its_oxygensensor.Status = status;
44              CheckEngineStatus();
45          }
46
47          public void setTemperatureSensor(PartStatus status) {
48              its_temperaturesensor.Status = status;
49              CheckEngineStatus();
50          }
51
52          public bool CheckEngineStatus() {
53              if (its_compressor.IsWorking && its_fuelpump.IsWorking &&
54                 its_oilpump.IsWorking && its_oxygensensor.IsWorking &&
55                 its_temperaturesensor.IsWorking) {
56                  its_status = PartStatus.WORKING;
57                  Console.WriteLine("All engine #" + its_engine_number + " components working properly.");
58              }
59              else {
60                  its_status = PartStatus.NOT_WORKING;
61                  Console.WriteLine("Engine #" + its_engine_number + " malfunction.");
62                  if (is_running) {
63                      Console.WriteLine("Engine #" + its_engine_number + " shutting down!");
64                      StopEngine();
65                  }
66              }
67
68              return its_status == PartStatus.WORKING ? true : false;
69          }
70
71          public void StartEngine() {
72              if (!is_running) {
73                  if (CheckEngineStatus()) {
74                      is_running = true;
75                      Console.WriteLine("Engine #" + its_engine_number + " is running!");
76                  }
77                  else {
78                      Console.WriteLine("There is a problem with an engine #" + its_engine_number
79                                        + " component. Engine cannot start.");
80                  }
81              }
82              else {
83                  Console.WriteLine("Engine #" + its_engine_number + " is already running!");
84              }
85          }
86
87          public void StopEngine() {
88              is_running = false;
89              Console.WriteLine("Engine #" + its_engine_number + " has been stopped!");
90          }
91
92      } // end class definition
93  } // end namespace
```

10.9 Compressor.cs

```
1   using System;
2
3   namespace EngineSimulation {
4
5       public class Compressor {
6           private int registered_engine_number = 0;
7           private PartStatus part_status;
8
9           public PartStatus Status {
10              get { return part_status; }
11              set { part_status = value; }
12          }
13
14          public int RegisteredEngineNumber {
15              get { return registered_engine_number; }
16              set { registered_engine_number = value; }
17          }
18
19          public bool IsWorking {
20              get {
21                  if (Status == PartStatus.WORKING) {
22                      return true;
23                  }
24                  return false;
```


© 2008 Rick Miller — All Rights Reserved

```
25              }
26          }
27
28          public Compressor(PartStatus status, int engine_number) {
29              RegisteredEngineNumber = engine_number;
30              Status = status;
31              Console.WriteLine("Compressor Created...");
32          }
33      } // end class definition
34
35  } // end namespace
```

10.10 FuelPump.cs

```
1   using System;
2
3   namespace EngineSimulation {
4
5       public class FuelPump {
6           private int registered_engine_number = 0;
7           private PartStatus part_status;
8
9           public PartStatus Status {
10              get { return part_status; }
11              set { part_status = value; }
12          }
13
14          public int RegisteredEngineNumber {
15              get { return registered_engine_number; }
16              set { registered_engine_number = value; }
17          }
18
19          public bool IsWorking {
20              get {
21                  if (Status == PartStatus.WORKING) {
22                      return true;
23                  }
24                  return false;
25              }
26          }
27
28          public FuelPump(PartStatus status, int engine_number) {
29              RegisteredEngineNumber = engine_number;
30              Status = status;
31              Console.WriteLine("FuelPump Created...");
32          }
33      } // end class definition
34
35  } // end namespace
```

10.11 OilPump.cs

```
1   using System;
2
3   namespace EngineSimulation {
4
5       public class OilPump {
6           private int registered_engine_number = 0;
7           private PartStatus part_status;
8
9           public PartStatus Status {
10              get { return part_status; }
11              set { part_status = value; }
12          }
13
14          public int RegisteredEngineNumber {
15              get { return registered_engine_number; }
16              set { registered_engine_number = value; }
17          }
18
19          public bool IsWorking {
20              get {
21                  if (Status == PartStatus.WORKING) {
22                      return true;
23                  }
24                  return false;
25              }
26          }
```


© 2008 Rick Miller — All Rights Reserved

```
27
28          public OilPump(PartStatus status, int engine_number) {
29              RegisteredEngineNumber = engine_number;
30              Status = status;
31              Console.WriteLine("OilPump Created...");
32          }
33      } // end class definition
34  } // end namespace
```

10.12 OxygenSensor.cs

```
1   using System;
2
3   namespace EngineSimulation {
4
5       public class OxygenSensor {
6           private int registered_engine_number = 0;
7           private PartStatus part_status;
8
9           public PartStatus Status {
10              get { return part_status; }
11              set { part_status = value; }
12          }
13
14          public int RegisteredEngineNumber {
15              get { return registered_engine_number; }
16              set { registered_engine_number = value; }
17          }
18
19          public bool IsWorking {
20              get {
21                  if (Status == PartStatus.WORKING) {
22                      return true;
23                  }
24                  return false;
25              }
26          }
27
28          public OxygenSensor(PartStatus status, int engine_number) {
29              RegisteredEngineNumber = engine_number;
30              Status = status;
31              Console.WriteLine("OxygenSensor Created...");
32          }
33      } // end class definition
34  } // end namespace
```

10.13 TemperatureSensor.cs

```
1   using System;
2
3   namespace EngineSimulation {
4
5       public class TemperatureSensor {
6           private int registered_engine_number = 0;
7           private PartStatus part_status;
8
9           public PartStatus Status {
10            get { return part_status; }
11            set { part_status = value; }
12          }
13
14          public bool IsWorking {
15            get {
16                if (Status == PartStatus.WORKING) {
17                    return true;
18                  }
19                  return false;
20              }
21          }
22
23          public int RegisteredEngineNumber {
24              get { return registered_engine_number; }
25              set { registered_engine_number = value; }
26          }
27
28          public TemperatureSensor(PartStatus status, int engine_number) {
29              RegisteredEngineNumber = engine_number;
30              Status = status;
```


© 2008 Rick Miller — All Rights Reserved

```
31              Console.WriteLine("TemperatureSensor Created...");
32          }
33      } // end class definition
34  } // end namespace
```

10.14 PartStatus..cs

```
1   using System;
2
3   namespace EngineSimulation {
4
5       public enum PartStatus { WORKING, NOT_WORKING }
6
7   }
```

10.15 EngineTester..c

```
1   using EngineSimulation;
2
3   public class EngineTester {
4     public static void Main(){
5      Engine e1 = new Engine(1);
6      e1.StartEngine();
7      e1.SetCompressorStatus(PartStatus.NOT_WORKING);
8      e1.StartEngine();
9      e1.SetCompressorStatus(PartStatus.WORKING);
10     e1.StartEngine();
11    }
12  }
```

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 UML. Physical complexity can be managed by your IDE or with an automated build tool such as Microsoft Build (MSBuild). You can also manage physical complexity on a small scale by organizing your source files into project directories and by compiling multiple files simultaneously using the `csc` 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.

A UML class diagram can be used to show aggregation associations between classes. A hollow diamond denotes simple aggregation and expresses a *uses* or *uses a* relationship between the whole and part classes. A solid diamond denotes composite aggregation and expresses a *contains* 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 classes in the engine simulation use the PartStatus enumeration.

Skill-Building Exercises

1. **Study The UML:** Obtain a reference on 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.

© 2008 Rick Miller — All Rights Reserved

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. **Discover Dependency Relationships:** Study the .NET Framework API. Write down at least five instances in which one class depends on the services of another class. Describe the nature of the dependency and explain the possible effects that changing the depended-upon class might have on the dependent class.

4. **Programming Exercise — Demonstrate Simple Aggregation:** Write a program that implements the simple aggregation shown in Figure 10-15.

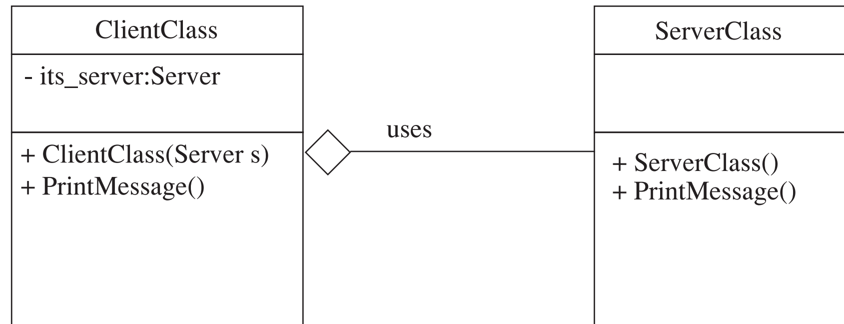


Figure 10-15: Simple Aggregation Class Diagram

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 10.1 through 10.3 for clues on how to implement this exercise.

5. **Programming Exercise — Demonstrate Composite Aggregation:** Write a program that implements the composite aggregation shown in Figure 10-16.

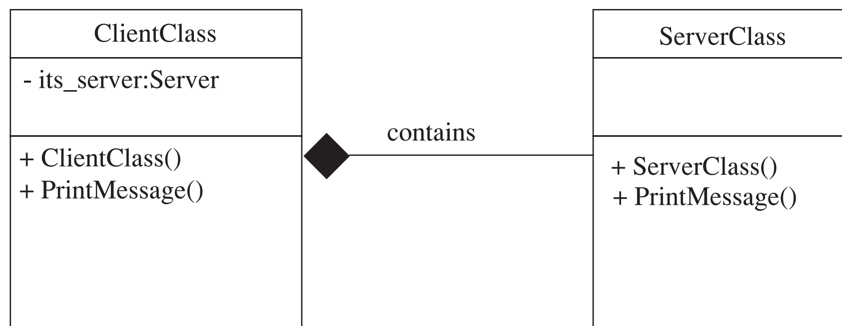


Figure 10-16: Composite Aggregation Class Diagram

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.

6. **Compile Multiple Source Files:** Find the engine simulator project source code presented in this chapter on the PulpFreePress C# For Artists' SupportSite®. [http://www.pulpfreepress.com] Use the `csc` compiler tool to compile all the source files in the project directory using the '*' wildcard character.

 © 2008 Rick Miller — All Rights Reserved

Suggested Projects

1. **Create Sequence Diagrams For 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 engine simulation project write a program that simulates an 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 an analysis and determine what parts your simulated car needs. Modify the 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 — Gasoline Pump Simulation:** Write a program that implements a simple gas pump system. Perform an 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 Space Telescope:** Write a program that controls the Hubble Space Telescope. Perform an 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 an 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 *contains* or *has a* relationship between two objects?

10. What is the purpose of a UML sequence diagram?

© 2008 Rick Miller — All Rights Reserved

11. A class built from other class types is referred to as a/an ________________________.

12. In this type of aggregation, part objects belong solely to the whole or containing class. Name the type of aggregation.

13. In this type of aggregation, part object lifetimes are not controlled by the whole or containing class. Name the type of aggregation.

14. 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

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

Sinan Si Alhir. *UML In A Nutshell: A Desktop Quick Reference*. O'Reilly and Associates, Inc., Sebastopol, CA. ISBN: 1-56592-448-7

Rick Miller. *Java For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-05-2

Notes

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Chapter 11

Pentax 67 / SMC Takumar 55/2.8 / Kodak Tri-X Professional

Fairview Park

Inheritance and Interfaces

Learning Objectives

- State the three essential purposes 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
- Define the following keywords: "sealed", "virtual," "override", "new", and "protected"
- Demonstrate your ability to use inheritance to create class hierarchies
- Demonstrate your ability to override base class methods in derived classes
- State the purpose of an interface
- Demonstrate your ability to create classes that implement interfaces
- State the definition of the term polymorphism
- Demonstrate your ability to write polymorphic code
- Express inheritance relationships using UML class diagrams

© 2008 Rick Miller — All Rights Reserved

Introduction

Inheritance is a powerful feature provided by modern object-oriented programming languages. The behavior provided or specified by one class can be adopted or extended by another class. Up to this point you have been using inheritance in every program you have written, although mostly this has been done for you behind the scenes by the C# compiler. For example, every user-defined class you create automatically inherits from the System.Object class.

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) by combining 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 *sealed* keyword to inhibit the inheritance mechanism. You will also learn how to use a Unified Modeling Language (UML) class diagram to show inheritance hierarchies.

By the time you complete this chapter, you will fully understand how to create an object-oriented C# program that exhibits *dynamic polymorphic behavior*. Most importantly, however, you will understand why dynamic polymorphic behavior is a desired object-oriented design objective.

This chapter also builds on the material presented in *Chapter 10 - Compositional Design*. The primary code example in this chapter 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 both *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 designing with inheritance, you create class hierarchies, where *base classes* that implement *generalized behavior* appear at or near the top of the hierarchy, and *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.

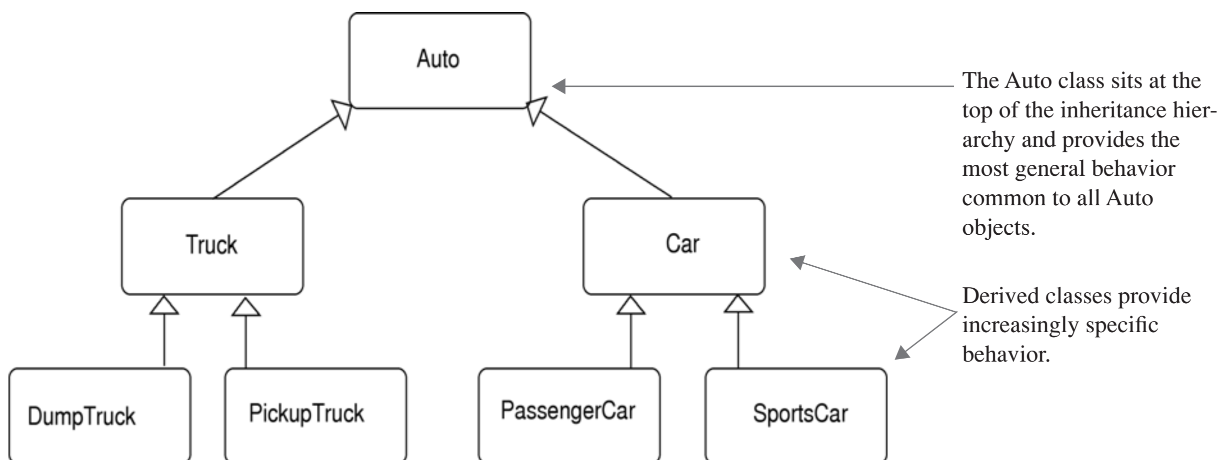


Figure 11-1: Inheritance Hierarchy Illustrating Generalized and Specialized Behavior

 © 2008 Rick Miller — All Rights Reserved

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 provide specialized behavior found only in Trucks and Car objects. 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 also implement the specialized behavior required of these class types. 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 .NET 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 .NET 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 C#.NET programming and in object-oriented programming in general.

The Relationship Between The Terms Type, Interface, and Class

Before moving on, it will help you to understand the relationship between the terms *type, interface* and *class*. C# is a strongly typed programming language. This means that when you write a program and wish to call a method on a particular object, the compiler 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 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 to compile your program.

Meaning Of The Term Interface

An *interface* is a construct that introduces a new data type and its set of authorized operations in the form of method, property, event, or indexer declarations. An interface member declaration provides a specification only and no implementation. Interfaces are discussed in more detail later in the chapter.

Meaning Of The Term Class

A *class* is a construct that introduces and defines a new data type. Like the interface, the class specifies a set of legal operations that can be performed 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, properties, events, or indexers. A class that provides definitions for all of its members 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 members, 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.

© 2008 Rick Miller — All Rights Reserved

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.

Class and interface constructs are each used to create new, user-defined data types. The interface construct specifies a set of authorized type operations and omits their behavior; the class construct specifies a set of authorized type operations and, optionally, their behavior as well. A class construct, like an interface, can omit the bodies of one or more of its members. Such members must be declared to be abstract. A class that declares one or more abstract members must be declared an abstract class. Abstract class objects cannot be created with the `new` operator.

Expressing Generalization And Specialization In The UML

Generalization and 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.

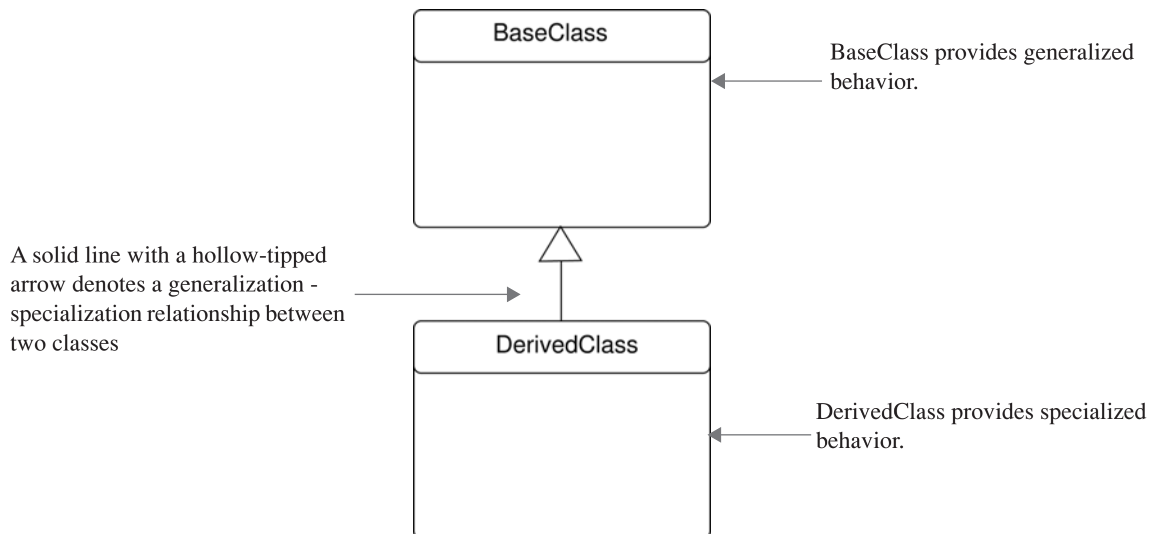


Figure 11-2: UML Class Diagram Showing DerivedClass Inheriting from BaseClass

Referring to Figure 11-2 — BaseClass acts as the *direct base class* to DerivedClass. Behavior provided by BaseClass is inherited by DerivedClass. The extent of BaseClass behavior that's inherited by DerivedClass is controlled by the use of the member access modifiers `public`, `protected`, `internal`, `protected internal`, and `private`. Generally speaking, base class members declared to be `public`, `protected`, `internal`, or `protected internal` are inherited by a derived class. A detailed discussion of how these access modifiers are used to control horizontal and vertical member access is presented later in this chapter. For now, however, let's take a look at an example program that implements the two classes shown in Figure 11-2.

© 2008 Rick Miller — All Rights Reserved

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 that 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, properties, and methods of BaseClass and DerivedClass is presented in Figure 11-3

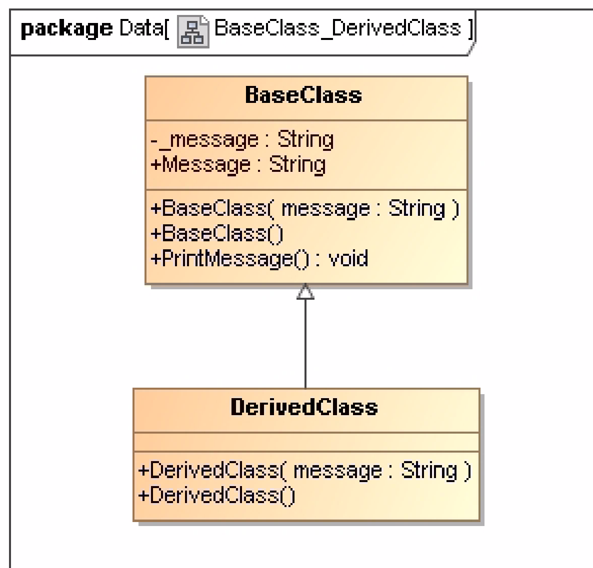


Figure 11-3: UML Diagram of BaseClass and DerivedClass Showing Fields, Properties, and Methods

Referring to Figure 11-3 — BaseClass contains one private field named _message which is of type String. It has one public property named Message. BaseClass has three public methods: two constructors and the PrintMessage() method. One of the constructors is a default 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() method and the Message property can be called on that object.

DerivedClass has only two constructors that are similar to the constructors found in BaseClass. It inherits the public members of BaseClass, which include the Message property and the PrintMessage() method. Let's now take a look at the source code for each class.

BaseClass Source Code

11.1 BaseClass.cs

```
1    using System;
2
3    public class BaseClass {
4      private String _message;
5
6      public String Message {
7        get { return _message; }
8        set { _message = value; }
9      }
10
11     public BaseClass(String message){
```

 © 2008 Rick Miller — All Rights Reserved

```
12        Console.WriteLine("BaseClass object created...");
13        Message = message;
14      }
15
16      public BaseClass():this("Default BaseClass message"){ }
17
18      public void PrintMessage(){
19        Console.WriteLine("BaseClass PrintMessage(): " + Message);
20      }
21    }
```

Referring to Example 11.1 — BaseClass is fairly simple. Its first constructor begins on line 11 and declares one string parameter named message. The _message field is set via the Message property. The default constructor begins on line 16. It calls the first constructor with the string literal "Default BaseClass message!" The PrintMessage() method begins on line 18. It simply prints the Message property to the console. A BaseClass object's message can be changed by setting its Message property.

Since the Message property and the PrintMessage() method each have a body, and are therefore defined, the BaseClass is considered a *concrete class*. This means that objects of type BaseClass can be created with the `new` operator.

Example 11.2 gives the code for DerivedClass.

DerivedClass Source Code

11.2 DerivedClass.cs

```
1     using System;
2
3     public class DerivedClass:BaseClass {
4
5       public DerivedClass(String message):base(message){
6         Console.WriteLine("DerivedClass object created...");
7       }
8
9       public DerivedClass():this("Default DerivedClass message"){ }
10    }
```

Referring to Example 11.2 — DerivedClass inherits the functionality of BaseClass by extending BaseClass. Note that on line 3, the name BaseClass follows the colon character ':'. DerivedClass itself provides only two constructors. The first constructor begins on line 5. It declares a string parameter named message. The first thing this constructor does is call the string parameter version of the BaseClass constructor using the base() method with the message parameter as an argument. Note how the call to base() follows the colon. The next thing the DerivedClass constructor does is print a short message to the console.

DerivedClass's default constructor begins on line 9. It calls its version of the string parameter constructor using the string literal *"Default DerivedClass message!"* as an argument to the this() call. This ultimately results in a call to the version of the BaseClass constructor that takes a string argument.

Let's now take a look at how these two classes can be used in a program.

DriverApplication Program

11.3 DriverApplication.cs

```
1     public class DriverApplication {
2       public static void Main(){
3         BaseClass b1 = new BaseClass();
4         BaseClass b2 = new DerivedClass();
5         DerivedClass d1 = new DerivedClass();
6
7         b1.PrintMessage();
8         b2.PrintMessage();
9         d1.PrintMessage();
10      }
11    }
```

The DriverApplication class tests the functionality of BaseClass and DerivedClass. The important thing to note in this example is which type of object is being declared and created on lines 3 through 5. Starting on line 3, a BaseClass reference named b1 is declared and initialized to point to a BaseClass object. On line 4, another BaseClass reference named b2 is declared and initialized to point to a DerivedClass object. On line 5, a DerivedClass reference named d1 is declared and initialized to point to a DerivedClass object. Note that a reference to a base class object can

© 2008 Rick Miller — All Rights Reserved

also point to a derived class object. Also note that this example only uses the default constructors to create each object. This results in the default message text being used upon the creation of each type of object.

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.

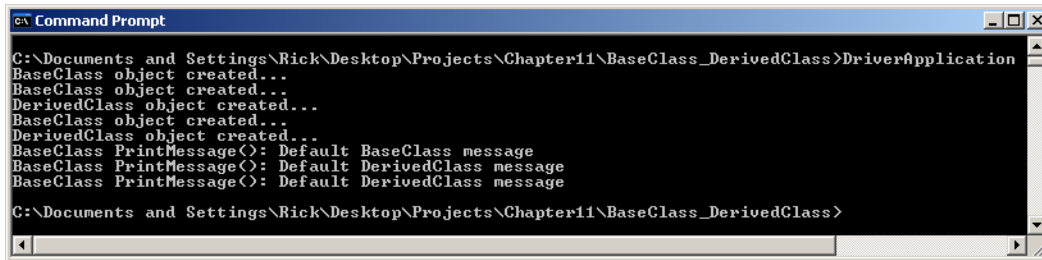


Figure 11-4: Results of Running Example 11.3

As you will notice from studying Figure 11-4, there are eight lines of program output that correspond to the creation of the three objects and the three PrintMessage() method calls on each reference b1, b2, and d1. 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. The value of the resulting text printed to the console shows 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. The PrintMessage() method is inherited by DerivedClass (*i.e.,* it is accessible to it) because it is declared public.

Finally, declaring 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 public, protected, internal, and protected internal members 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 realistic example of inheritance. This example uses 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 and properties. The Student class extends the functionality of Person and provides a small bit of specialized functionality of its own in the form of the StudentNumber and Major properties.

Since the Student class participates in an *is-a* relationship with class Person, a Student object can be used wherever a Person object is called for in your source code. However, now you must be keenly aware of the specialized

© 2008 Rick Miller — All Rights Reserved

behavior provided by the Student class, as you will soon see when you examine and run the driver application program for this example.

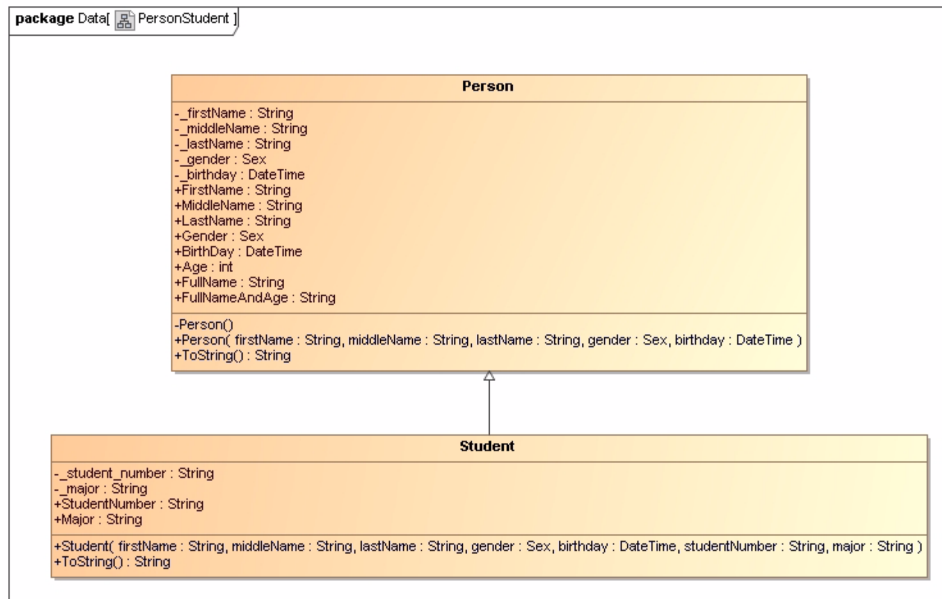


Figure 11-5: UML Diagram Showing Student Class Inheritance Hierarchy

Person - Student Source Code

11.4 Person.cs

```
1    using System;
2
3    public class Person {
4
5      //enumeration
6      public enum Sex {MALE, FEMALE};
7
8      // private instance fields
9      private String   _firstName;
10     private String   _middleName;
11     private String   _lastName;
12     private Sex      _gender;
13     private DateTime _birthday;
14
15     //private default constructor
16     private Person(){}
17
18     public Person(String firstName, String middleName, String lastName,
19                   Sex gender, DateTime birthday){
20        FirstName = firstName;
21        MiddleName = middleName;
22        LastName = lastName;
23        Gender = gender;
24        BirthDay = birthday;
25     }
26
27     // public properties
28     public String FirstName {
29       get { return _firstName; }
30       set { _firstName = value; }
31     }
32
33     public String MiddleName {
34       get { return _middleName; }
35       set { _middleName = value; }
36     }
37
38     public String LastName {
```

 © 2008 Rick Miller — All Rights Reserved

```
39        get { return _lastName; }
40        set { _lastName = value; }
41      }
42
43      public Sex Gender {
44        get { return _gender; }
45        set { _gender = value; }
46      }
47
48      public DateTime BirthDay {
49        get { return _birthday; }
50        set { _birthday = value; }
51      }
52
53       public int Age {
54         get {
55           int years = DateTime.Now.Year - _birthday.Year;
56           int adjustment = 0;
57             if(DateTime.Now.Month < _birthday.Month){
58               adjustment = 1;
59             }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
60                       adjustment = 1;
61                   }
62           return years - adjustment;
63         }
64       }
65
66      public String FullName {
67        get { return FirstName + " " + MiddleName + " " + LastName; }
68      }
69
70      public String FullNameAndAge {
71        get { return FullName + " " + Age; }
72      }
73
74      public override String ToString(){
75        return FullName + " is a " + Gender + " who is " + Age + " years old.";
76      }
77    } // end Person class
```

The Person class code is unchanged from Chapter 9.

11.5 Student.cs

```
1     using System;
2
3     public class Student:Person {
4       private String _student_number;
5       private String _major;
6
7       public String StudentNumber {
8         get { return _student_number; }
9         set { _student_number = value; }
10      }
11
12      public String Major {
13        get { return _major; }
14        set { _major = value; }
15      }
16
17      public Student(String firstName, String middleName, String lastName,
18                     Sex gender, DateTime birthday, String studentNumber,
19                     String major):base(firstName, middleName, lastName, gender, birthday) {
20           StudentNumber = studentNumber;
21           Major = major;
22      }
23
24      public override String ToString(){
25        return (base.ToString() + " Student Number: " + StudentNumber + " Major: " + Major);
26      }
27
28    } // end Student class definition
```

Referring to Example 11.5 — the Student class extends Person and implements specialized behavior in the form of the StudentNumber and Major properties. The Student class has one constructor. With the exception of the last two parameters, studentNumber and major, the parameters are those required by the Person class. Note how the required person constructor arguments are used in the call to base() on line 19. The parameters studentNumber and major are then used on lines 20 and 21, respectively to set a Student object's StudentNumber and Major properties.

© 2008 Rick Miller — All Rights Reserved

The Student class also overrides the ToString() method, which begins on line 24. Note how the Person class's version of ToString() is called via base.ToString(). The required additional Student information is appended to this string and returned.

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.cs

```
1    using System;
2
3    public class PersonStudentTestApp {
4      public static void Main(){
5        Person p1 = new Person("Ulysses", "S", "Grant", Person.Sex.MALE, new DateTime(1822, 04, 22));
6        Person p2 = new Student("Steven", "Jay", "Jones", Person.Sex.MALE, new DateTime(1986, 03, 21),
7                                "1234564", "Finance");
8        Student s1 = new Student("Virginia", "LeAnn", "Mattson", Person.Sex.FEMALE,
9                                 new DateTime(1973, 09, 14), "8798765", "Computer Science");
10       Console.WriteLine(p1);
11       Console.WriteLine(p2);
12       Console.WriteLine(s1);
13
14       // p2.Major = "Criminal Justice"; // ERROR: p2 is a Person reference
15       s1.Major = "Physics";
16
17       Console.WriteLine("---------------------------------------------");
18       Console.WriteLine(p2);
19       Console.WriteLine(s1);
20     }
21   }
```

Referring to Example 11.6 — this program is similar in structure to Example 11-3 in that it declares three references and shows you the effects of accessing methods and properties via those references. A Person reference named p1 is declared on line 5 and initialized to point to a Person object. On line 6, another Person reference named p2 is declared and initialized to point to a Student object. On line 8, a Student reference is declared and initialized to point to a Student object. On lines 10, 11, and 12, each object's information is printed to the console.

Line 14 is commented out. This line, if you were to try to compile it, will cause a compiler error because an attempt is made to set the Major property on a Student object via a Person 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 set the Major property on a Student object?" You can, but p2 is a Person type reference, which means that the compiler is enforcing the interface defined by the Person class. Remember the "C# 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 15, the Major property is set on a Student object via a Student reference. Finally, on lines 18 and 19, the information for references p2 and s1 is printed to the console. Figure 11-6 gives the results of running Example 11.6.

```
Command Prompt
C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\Person_Student>PersonStudentTestApp
Ulysses S Grant is a MALE who is 185 years old.
Steven Jay Jones is a MALE who is 21 years old. Student Number: 1234564 Major: Finance
Virginia LeAnn Mattson is a FEMALE who is 34 years old. Student Number: 8798765 Major: Computer Science
---------------------------------------------
Steven Jay Jones is a MALE who is 21 years old. Student Number: 1234564 Major: Finance
Virginia LeAnn Mattson is a FEMALE who is 34 years old. Student Number: 8798765 Major: Physics

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\Person_Student>_
```

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 14. Example 11.7 gives the modified version of PersonStudentTestApp.cs.

 © 2008 Rick Miller — All Rights Reserved

11.7 PersonStudentTestApp.cs (Mod 1)

```
1    using System;
2
3    public class PersonStudentTestApp {
4      public static void Main(){
5        Person p1 = new Person("Ulysses", "S", "Grant", Person.Sex.MALE, new DateTime(1822, 04, 22));
6        Person p2 = new Student("Steven", "Jay", "Jones", Person.Sex.MALE, new DateTime(1986, 03, 21),
7                                "1234564", "Finance");
8        Student s1 = new Student("Virginia", "LeAnn", "Mattson", Person.Sex.FEMALE, new DateTime(1973, 09, 14),
9                                 "8798765", "Computer Science");
10       Console.WriteLine(p1);
11       Console.WriteLine(p2);
12       Console.WriteLine(s1);
13
14       ((Student)p2).Major = "Criminal Justice"; // OK - Person reference is cast to type Student
15       s1.Major = "Physics";
16
17       Console.WriteLine("---------------------------------------------");
18       Console.WriteLine(p2);
19       Console.WriteLine(s1);
20     }
21   }
```

Referring to Example 11.7 — notice on line 14 that the compiler has been instructed to treat the p2 reference as though it were a Student type reference. This form of explicit *type coercion* is called *casting*. Casting only works if the object the reference actually points to is of the proper type. In other words, you can cast p2 to a Student type because it points to a Student object. However, you could not cast the p1 reference to Student since it actually points to a Person object. Figure 11-7 shows the results of running Example 11.7.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\Person_Student_casting>PersonStudentTestApp
Ulysses S Grant is a MALE who is 185 years old.
Steven Jay Jones is a MALE who is 21 years old. Student Number: 1234564 Major: Finance
Virginia LeAnn Mattson is a FEMALE who is 34 years old. Student Number: 8798765 Major: Computer Science
---------------------------------------------
Steven Jay Jones is a MALE who is 21 years old. Student Number: 1234564 Major: Criminal Justice
Virginia LeAnn Mattson is a FEMALE who is 34 years old. Student Number: 8798765 Major: Physics

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\Person_Student_casting>
```

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 try to 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 its own specialized behavior.

Reference variables have an associated type. Method calls to an object pointed to by a reference will succeed without casting as long as the reference type supports the method you are trying to call. Casting forces, or coerces, the compiler into treating 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, use casting sparingly. Also, casting only works if the object really is of the type you are casting it to.

© 2008 Rick Miller — All Rights Reserved

Overriding Base Class Methods

So far you have only seen examples of inheritance in which the derived class fully accepted the behavior provided by its base class. This section shows 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 need to redefine 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 and be declared with the keyword `override`. You also need to add the keyword `virtual` to the base class method declaration. By using the `virtual`/`override` keyword pair, you can achieve polymorphic behavior.

Let's take a look at a simple example. Figure 11-8 gives a UML class diagram for the revised BaseClass and DerivedClass classes.

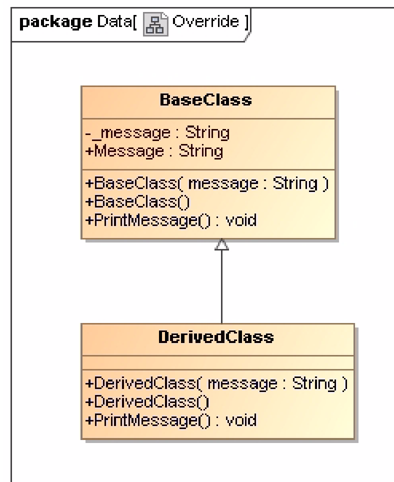


Figure 11-8: UML Class Diagram For BaseClass & DerivedClass

Referring to Figure 11-8 — notice that DerivedClass now has a public method named PrintMessage(). BaseClass has been modified by adding the keyword `virtual` to the declaration of its PrintMessage() method, which is not shown in the diagram. Example 11.8 gives the source code for the modified version of BaseClass.

11.8 BaseClass.cs (Mod 1)

```
1    using System;
2
3    public class BaseClass {
4      private String _message;
5
6      public String Message {
7        get { return _message; }
8        set { _message = value; }
9      }
10
11     public BaseClass(String message){
12       Console.WriteLine("BaseClass object created...");
13       Message = message;
14     }
15
16     public BaseClass():this("Default BaseClass message"){ }
17
18     public virtual void PrintMessage(){
19       Console.WriteLine("BaseClass PrintMessage(): " + Message);
20     }
21   }
```

Referring to Example 11.8 — the only change to BaseClass is the addition of the `virtual` keyword to the definition of the PrintMessage() method. The `virtual` keyword enables the PrintMessage() method to be overridden in DerivedClass. If you omit the `virtual` keyword from a base class method or other overrideable member, then you will get a compiler error if you attempt to override that member in a derived class.

Example 11.9 gives the code for the modified version of DerivedClass.

 © 2008 Rick Miller — All Rights Reserved

11.9 DerivedClass.cs (Mod 1)

```
1    using System;
2
3    public class DerivedClass:BaseClass {
4
5      public DerivedClass(String message):base(message){
6
7        Console.WriteLine("DerivedClass object created...");
8      }
9
10     public DerivedClass():this("Default DerivedClass message"){ }
11
12     public override void PrintMessage(){
13       Console.WriteLine("DerivedClass PrintMessage(): " + Message);
14     }
15   }
```

Referring to Example 11.9 —the DerivedClass's version of PrintMessage() on line 12 overrides the BaseClass version. Note the use of the `override` keyword in the PrintMessage() method definition. 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 the modified versions of BaseClass and DerivedClass.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\BaseClass_DerivedClass_Override>DriverApplication
BaseClass object created...
BaseClass object created...
DerivedClass object created...
BaseClass object created...
DerivedClass object created...
BaseClass PrintMessage(): Default BaseClass message
DerivedClass PrintMessage(): Default DerivedClass message
DerivedClass PrintMessage(): Default DerivedClass message

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\BaseClass_DerivedClass_Override>_
```

Figure 11-9: Results of Running Example 11.3 with Modified Versions of BaseClass and 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 b1 reference points to a BaseClass object. The second message is different, though. Why is this so? The b2 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, b2, points to a derived class object. You call a method provided by the base class interface via the base class reference, but is overridden in the derived class and, voila, you have polymorphic behavior. Pretty cool, huh?

Quick Review

Derived classes *override* base class behavior by providing *overriding methods*. An *overriding method* is a method in a derived class that has the same signature as the base class method it is intending to override. Use the `virtual` keyword to declare an overrideable base class method. Use the `override` keyword to define an overriding derived class method. Overriding methods can be called polymorphically via a base class reference that points to a derived class object.

Abstract Methods and 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 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.

 © 2008 Rick Miller — All Rights Reserved

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 constitute the upper tier of your inheritance hierarchy. The upper tier of an inheritance hierarchy is where you expect to find specifications for the general behavior inherited by derived classes, which appear in the lower tier 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*, rather than *modified*, to accommodate new functionality. This design technique injects a good dose of stability into your application architecture. This and other advanced object-oriented design techniques are discussed in more detail in Chapter 23.

Expressing Abstract Base Classes In UML

Figure 11-10 shows a UML diagram that contains an abstract base class named AbstractClass.

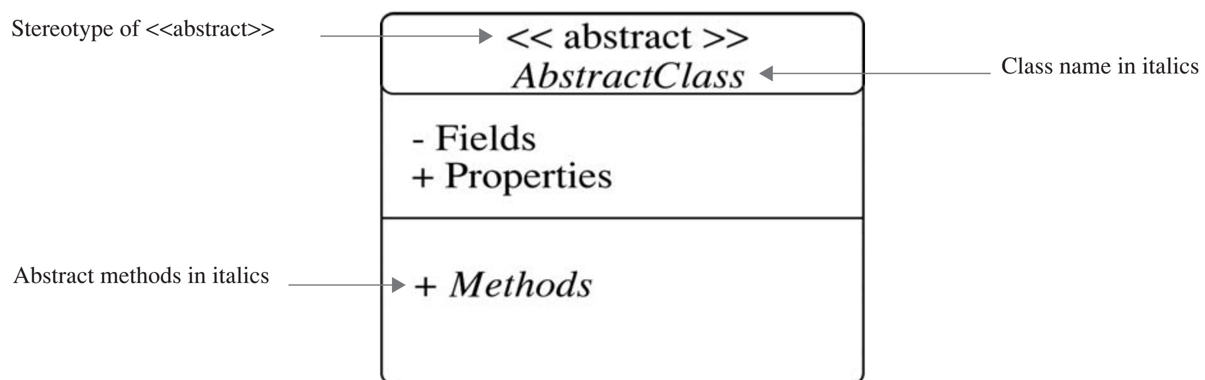


Figure 11-10: Expressing an Abstract Class in the UML

Referring to Figure 11-10 — the stereotype <<abstract>> is optional, but if your draw your UML diagrams by hand, it's hard to write in italics, so, this notation comes in handy. Abstract classes can have the same kinds of members as normal classes, but abstract members 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:

Referring to Figure 11-11 — AbstractClass has two methods and one property. The first method is its default constructor and it is not abstract. (Remember, constructors cannot be abstract.) The next method, PrintMessage(), is shown in italics and is therefore an abstract method. The Message property is also abstract in this example but a limitation in MagicDraw prevents it from being displayed in italics, otherwise, MagicDraw is a fine UML design tool.

DerivedClass inherits from AbstractClass. Since AbstractClass's PrintMessage() method is abstract and has no implementation, DerivedClass must provide an implementation for it. DerivedClass's PrintMessage() method is in plain font, indicating it has an implementation.

Now, if for some reason, you as a designer decided to create a class that inherited from DerivedClass, and you defer the implementation of the PrintMessage() method 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 the two-tiered inheritance hierarchy I have used here in this simple example.

Let's now take a look at the code for these two classes. Example 11.10 gives the code for AbstractClass.

 © 2008 Rick Miller — All Rights Reserved

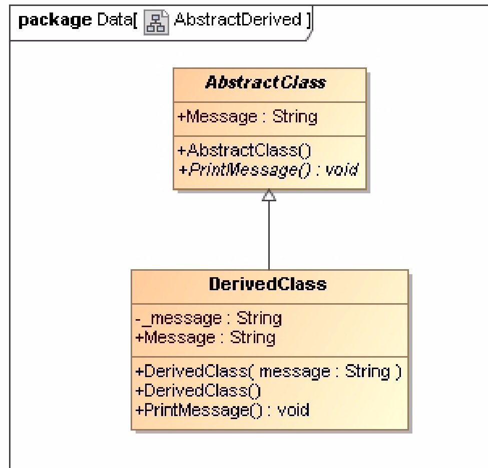


Figure 11-11: UML Class Diagram Showing the AbstractClass and DerivedClass Inheritance Hierarchy

11.10 AbstractClass.cs

```
1    using System;
2
3    public abstract class AbstractClass {
4
5      public abstract String Message {
6        get;
7        set;
8      }
9
10     public AbstractClass(){
11       Console.WriteLine("AbstractClass object created...");
12     }
13
14     public abstract void PrintMessage();
15   }
```

Referring to Example 11.10 — the important point to note is that on line 3 the keyword `abstract` indicates that this is an abstract class definition. The `abstract` keyword is also used on lines 5 and 14 in the Message property definition and the PrintMessage() method declaration. An abstract member is implicitly virtual, so you don't need to add the `virtual` keyword to an abstract member definition. In fact, doing so will produce a compiler warning. Also note how the Message property's `get` and `set` accessors are terminated with a semicolon, indicating they have no implementation. (*i.e.,* No curly braces, no body, no implementation.) The same holds true for the PrintMessage() method. Example 11.11 gives the code for DerivedClass.

11.11 DerivedClass.cs

```
1    using System;
2
3    public class DerivedClass:AbstractClass {
4      private String _message;
5
6      public override String Message {
7         get { return _message; }
8         set { _message = value; }
9      }
10
11     public DerivedClass(String message){
12       Message = message;
13       Console.WriteLine("DerivedClass object created...");
14     }
15
16     public DerivedClass():this("Default DerivedClass message"){ }
17
18     public override void PrintMessage(){
19       Console.WriteLine("DerivedClass PrintMessage(): " + Message);
20     }
21   }
```

Referring to Example 11.11 — DerivedClass extends AbstractClass. DerivedClass provides an implementation for each of AbstractClass's abstract members. Let's take a look now at the test driver program that exercises these two classes.

© 2008 Rick Miller — All Rights Reserved

11.12 DriverApplication.cs

```
1    public class DriverApplication {
2      public static void Main(){
3        AbstractClass a1 = new DerivedClass(); // preferred combination
4        DerivedClass  d1 = new DerivedClass();
5        a1.PrintMessage();
6        d1.PrintMessage();
7        a1.Message = "New Message";
8        d1.Message = "Another Message";
9        a1.PrintMessage();
10       d1.PrintMessage();
11     }
12   }
```

Referring to Example 11.12 — remember, you cannot directly instantiate an abstract class object. On line 3, a reference to an AbstractClass type object named a1 is declared and initialized to point to a DerivedClass object. On line 4, a reference to a DerivedClass type object named d1 is declared and initialized to point to a DerivedClass object.

The comment on line 3 that says “preferred combination” means that the abstract type reference pointing to the derived class object is the preferred combination in an object-oriented program. Remember, your goal is to write code that works the same regardless of what type of object a reference points to. The abstract class serves as a specification for behavior. As long as it points to an object that implements the specified behavior, things should work fine.

Figure 11-12 shows the results of running Example 11.12.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\AbstractDerived>driverapplication
AbstractClass object created...
DerivedClass object created...
AbstractClass object created...
DerivedClass object created...
DerivedClass PrintMessage(): Default DerivedClass message
DerivedClass PrintMessage(): Default DerivedClass message
DerivedClass PrintMessage(): New Message
DerivedClass PrintMessage(): Another Message

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\AbstractDerived>_
```

Figure 11-12: Results of Running Example 11.12

Quick Review

An *abstract member* is a member that omits its body and has no implementation behavior. A class that declares one or more abstract members must be declared to be `abstract`.

The primary purpose of an *abstract class* is to provide a specification for behavior whose implementation is 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 construct that functions like an implicit abstract class. In C#, a derived class can extend the behavior of only one class, but it can implement as many interfaces as it requires. Interfaces themselves can inherit (*i.e.,* extend) from multiple interfaces.

The Purpose Of Interfaces

The purpose of an interface is to provide a specification for behavior in the form of abstract properties, methods, events, and indexers. An interface declaration introduces a new data type, just as class declarations and definitions do.

Authorized Interface Members

C# interfaces can only contain four types of members. These include:

 © 2008 Rick Miller — All Rights Reserved

- Properties — Implicitly public and abstract.
- Methods — Implicitly public and abstract.
- Events — Implicitly public and abstract.
- Indexers — Implicitly public and abstract.

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 abstract with the `abstract` keyword.	Is implicitly abstract.
Can contain abstract and concrete members.	Can only contain abstract members. All members in an interface are implicitly public and abstract.
Can contain fields, constants, and static members.	Can only contain declarations for properties, methods, events, and indexers.
Can contain nested class and interface declarations.	Can only contain declarations for properties, methods, events, and indexers.
Can extend one class and 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.

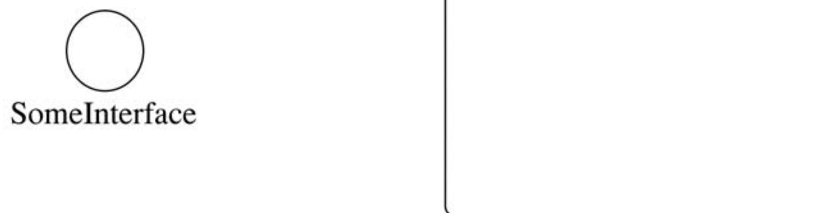


Figure 11-13: Two Types of UML Interface Diagrams

Referring to Figure 11-13 — one way to show an interface in UML is by using a 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 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* that interface. Interface *realization* is expressed in UML in two distinct forms: 1) the *simple form* in which the circle represents the interface and is combined with an association line to create a *lollipop diagram*, or 2) the *expanded form* in which an ordinary class diagram represents the interface. Figure 11-14 illustrates the use of the lollipop diagram to convey the simple form of realization. Figure 11-15 shows an example of the expanded form of realization.

© 2008 Rick Miller — All Rights Reserved

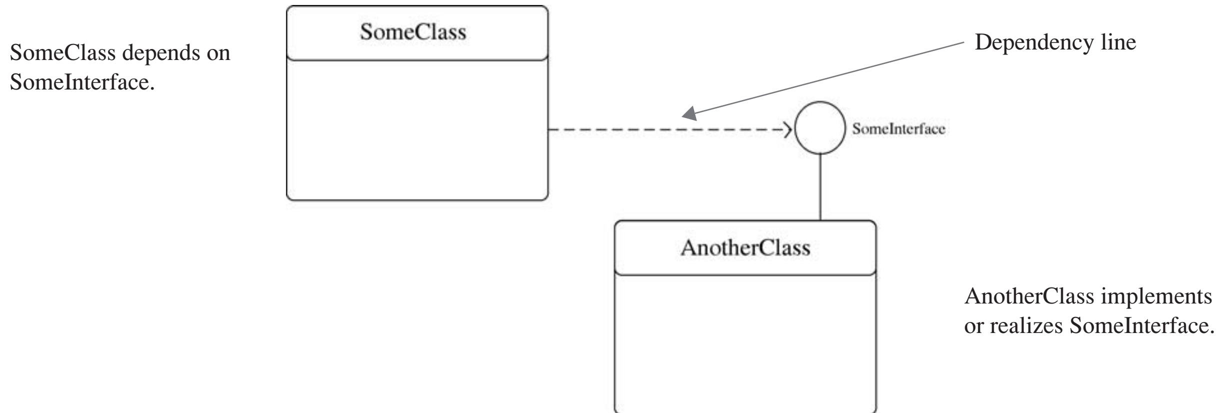


Figure 11-14: UML Diagram Showing the Simple Form of Realization

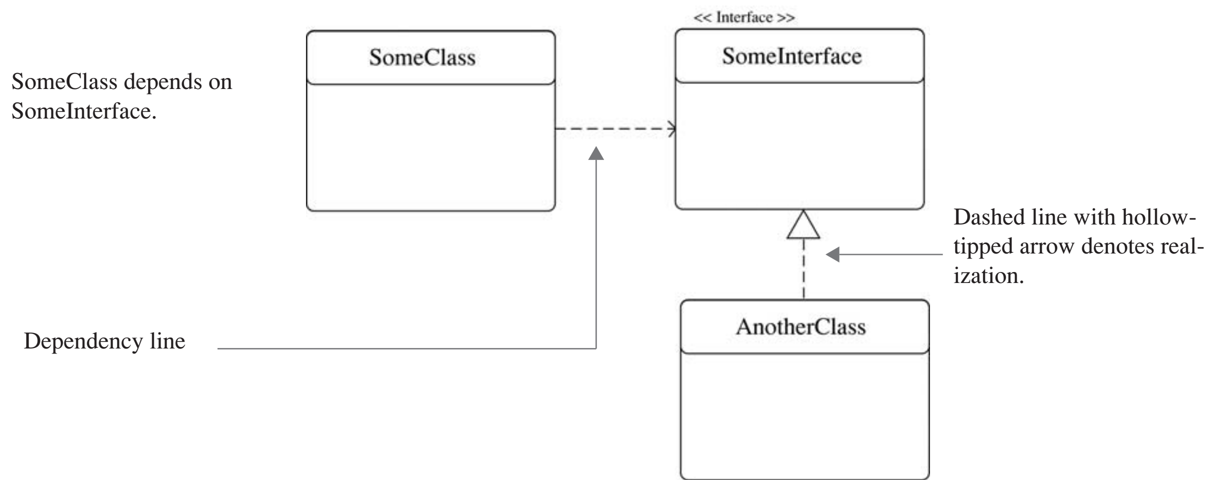


Figure 11-15: UML Diagram Showing the Expanded Form of Realization

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 IMessagePrinter interface and a class named MessagePrinter that implements the IMessagePrinter interface. The source code for these two classes is given in Examples 11.13 and 11.14.

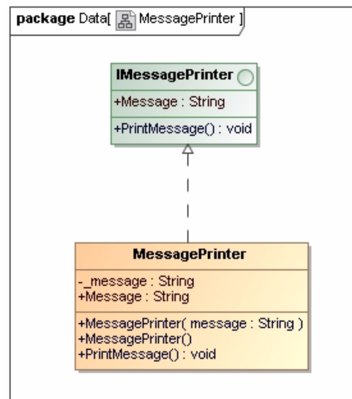


Figure 11-16: UML Diagram Showing the MessagePrinter Class Implementing the IMessagePrinter Interface

© 2008 Rick Miller — All Rights Reserved

11.13 IMessagePrinter.cs

```
1    using System;
2
3    public interface IMessagePrinter {
4      String Message {
5        get;
6        set;
7      }
8
9      void PrintMessage();
10   }
```

11.14 MessagePrinter.cs

```
1    using System;
2
3    public class MessagePrinter:IMessagePrinter {
4      private String _message;
5
6      public String Message {
7         get { return _message; }
8         set { _message = value; }
9      }
10
11     public MessagePrinter(String message){
12       Message = message;
13       Console.WriteLine("MessagePrinter object created...");
14     }
15
16     public MessagePrinter():this("Default MessagePrinter message"){ }
17
18     public void PrintMessage(){
19       Console.WriteLine("MessagePrinter PrintMessage(): " + Message);
20     }
21   }
```

11.15 DriverApplication.cs

```
1    public class DriverApplication {
2      public static void Main(){
3        IMessagePrinter i1 = new MessagePrinter();
4        MessagePrinter  m1 = new MessagePrinter();
5        i1.PrintMessage();
6        m1.PrintMessage();
7        i1.Message = "New Message";
8        m1.Message = "Another Message";
9        i1.PrintMessage();
10       m1.PrintMessage();
11     }
12   }
```

As you can see from Example 11.13, the IMessagePrinter interface is short and simple. All it does is declare two interface members: the Message property and the PrintMessage() method. The implementation of these interface members is left to any class that implements the IMessagePrinter interface, as the MessagePrinter class does in Example 11.14.

Example 11.15 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 i1. 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 object members you can access via the interface-type reference are those members specified by the interface. You could of course cast to a different type if required, as long as the object implements that type's interface, but strive to minimize the need to cast in this manner.

Figure 11-17 gives the results of running Example 11.15.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\Interfaces>driverapplication
MessagePrinter object created...
MessagePrinter object created...
MessagePrinter PrintMessage(): Default MessagePrinter message
MessagePrinter PrintMessage(): Default MessagePrinter message
MessagePrinter PrintMessage(): New Message
MessagePrinter PrintMessage(): Another Message

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\Interfaces>
```

Figure 11-17: Results of Running Example 11.15

© 2008 Rick Miller — All Rights Reserved

Quick Review

The purpose of an *interface* is to specify behavior. Interfaces can have four types of members: 1) properties, 2) methods, 3) events, and 4) indexers. Classes can inherit from or extend only one other class, but they can implement as many interfaces as required. Interfaces can extend as many interfaces as necessary.

Controlling Horizontal And Vertical Access

The term *horizontal access* describes the level of access an object of one type has to the members of another type. I discussed this topic 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 access modifiers `public`, `protected`, `private`, `internal`, and `protected internal`.

The default class member access is private. That is, when you omit an explicit access modifier from the definition of a class member, the member's accessibility is set to private by default. Conversely, interface members are public by default. A derived class does not have access to a base class's private members. (*i.e.,* Private members are not inherited.)

Derived classes have access to their base class's public, protected, internal, and protected internal members. (*i.e.,* These members are inherited by the derived class.)

The most frequently used access modifiers are `private`, `public`, and `protected`. As a rule of thumb you will declare a class's fields and one of more of its methods to be private. You saw an example of this already with private fields and private default constructors. Utility methods meant to be used only by their containing class are usually declared to be private as well.

If you want a member to be inherited by derived classes (*i.e.,* accessible vertically) but not accessible horizontally by other classes or code, declare it to be `protected`. If you want a member to be both horizontally and vertically accessible to all code within an assembly but only vertically accessible to derived classes outside the assembly, declare it to be `protected internal`.

Quick Review

Use the access modifiers `private`, `protected`, `public`, `internal`, and `protected internal` to control horizontal and vertical member access.

Sealed Classes And Methods

Sometimes you want to prevent classes from being extended or individual methods of a particular class from being overridden. Use the keyword `sealed` 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. Use the `sealed` keyword in conjunction with the `override` keyword. In other words, a sealed method is an overridden method that you want to prevent from being further overridden in the future.

You cannot use the keyword `sealed` in combination with the keyword `abstract` for obvious reasons.

Quick Review

Use the `sealed` keyword to stop the inheritance mechanism or prevent base class methods from being overridden in derived classes.

© 2008 Rick Miller — All Rights Reserved

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 members 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 at 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 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 *typeof* 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 specified by a base class or interface and manipulating their derived class objects via those operations. 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 11-18 gives the UML diagram. Referring to Figure 11-18 — the Employee class extends Person and implements the IPayable interface. However, in this example, as you will see later, the implementation of the Pay() method specified in the IPayable interface is deferred by the Employee class to its derived classes. It does this by mapping the IPayable.Pay() method to an abstract method, which means the Employee class now becomes an abstract class.

The HourlyEmployee and SalariedEmployee classes extend the functionality of Employee. Each of these classes will implement the Pay() method in its 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 members you can access on these objects via a Person reference without casting are those public members specified by the Person class.

Another approach would be to declare an array of IPayable references. Then again, you could initialize each IPayable reference to point to either an HourlyEmployee object or a SalariedEmployee object. Now the only members of each object that you can access 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 access any member specified by Person, IPayable, and Employee. This is the approach taken in the EmployeeTestApp program listed in Example 11.20.

The code for each of these classes (except Person class, which was shown earlier in the chapter) along with the EmployeeTestApp class is given in Examples 11.16 through 11.20.

 © 2008 Rick Miller — All Rights Reserved

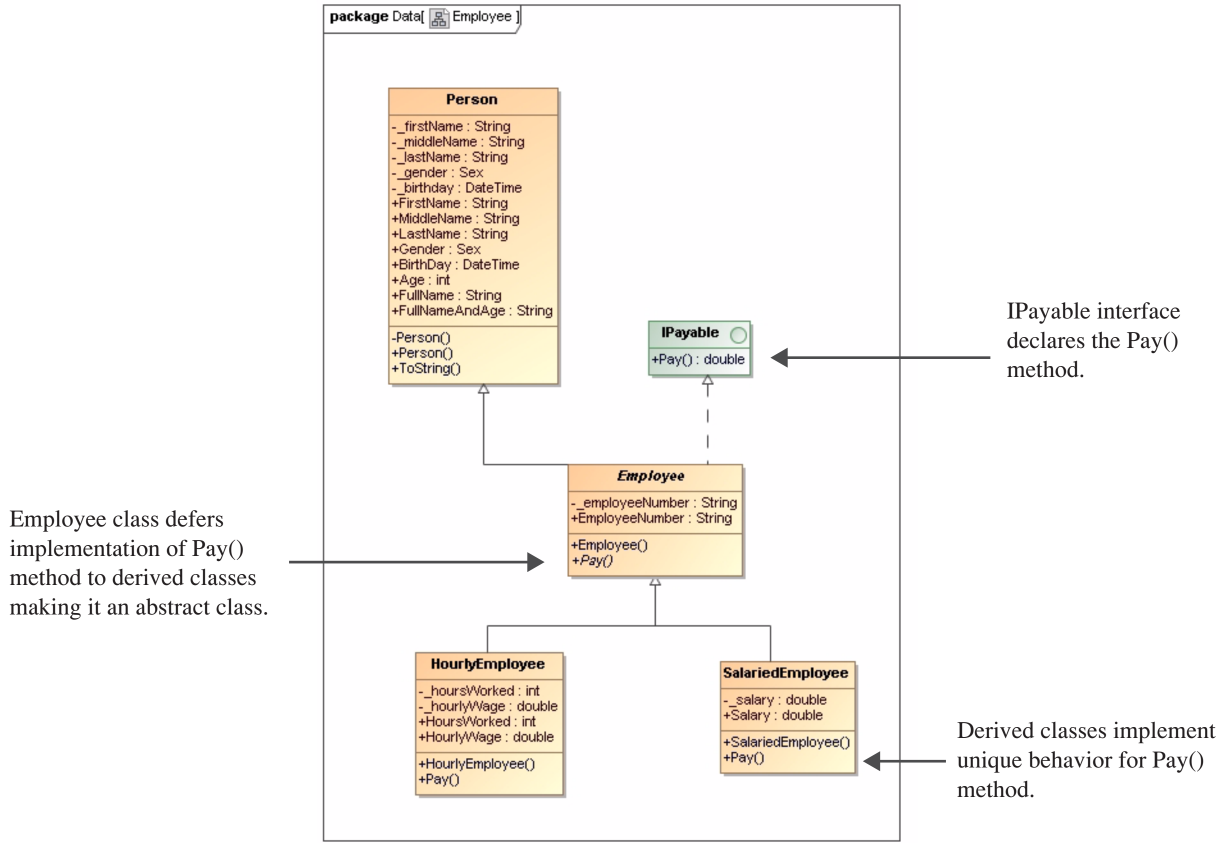


Figure 11-18: Employee Class Inheritance Hierarchy

11.16 IPayable.cs

```
1     using System;
2
3     public interface IPayable {
4        double Pay();
5     }
```

11.17 Employee.cs

```
1     using System;
2
3     /**************************************************
4     * The Employee class entends Person and implements
5     * IPayable, but since it defers the actual
6     * implementation of IPayable's Pay() method
7     * to derived classes it must be declared an
8     * abstract class.
9     **************************************************/
10
11    public abstract class Employee : Person, IPayable {
12       private String _employeeNumber;
13
14       public String EmployeeNumber {
15          get { return _employeeNumber; }
16          set { _employeeNumber = value; }
17       }
18
19       public Employee(String firstName, String middleName, String lastName,
20                      Sex gender, DateTime birthday, String employeeNumber):
21                      base(firstName, middleName, lastName, gender, birthday){
22         EmployeeNumber = employeeNumber;
23       }
24
25
26       public abstract double Pay(); // map IPayable.Pay() to an abstract method
27                                     // and defer implementation
28
29    } // end Employee class definition
```

 © 2008 Rick Miller — All Rights Reserved

11.18 HourlyEmployee.cs

```
1    using System;
2
3    public class HourlyEmployee : Employee {
4
5      private int _hoursWorked;
6      private double _hourlyWage;
7
8      public int HoursWorked {
9        get { return _hoursWorked; }
10       set { _hoursWorked = value; }
11     }
12
13     public double HourlyWage {
14       get { return _hourlyWage; }
15       set { _hourlyWage = value; }
16     }
17
18     public HourlyEmployee(String firstName, String middleName, String lastName,
19                 Sex gender, DateTime birthday, String employeeNumber, int hoursWorked,
20                 double hourlyWage): base(firstName, middleName, lastName, gender, birthday,
21                 employeeNumber){
22       HoursWorked = hoursWorked;
23       HourlyWage = hourlyWage;
24     }
25
26     public override double Pay(){
27       return HoursWorked * HourlyWage;
28     }
29
30   }
```

11.19 SalariedEmployee.cs

```
1    using System;
2
3    public class SalariedEmployee : Employee {
4
5      private double _salary;
6
7      public double Salary {
8        get { return _salary; }
9        set { _salary = value; }
10     }
11
12     public SalariedEmployee(String firstName, String middleName, String lastName,
13                 Sex gender, DateTime birthday, String employeeNumber, double salary):
14                 base(firstName, middleName, lastName, gender, birthday, employeeNumber){
15       Salary = salary;
16
17     }
18
19     public override double Pay(){
20       return Salary/24;;
21     }
22
23   }
```

11.20 EmployeeTestApp.cs

```
1    using System;
2
3    public class EmployeeTestApp {
4      public static void Main(){
5        Employee[] employees = new Employee[4];
6
7        employees[0] = new HourlyEmployee("Rick", "W", "Miller", Person.Sex.MALE,
8                                          new DateTime(1964,02,02), "11111111", 80, 17.00);
9
10       employees[1] = new SalariedEmployee("Steve", "J", "Jones", Person.Sex.MALE,
11                                         new DateTime(1975,08,09), "22222222", 130000.00);
12
13       employees[2] = new HourlyEmployee("Bob", "E", "Evans", Person.Sex.MALE,
14                                         new DateTime(1956,12,23), "33333333", 80, 25.00);
15
16       employees[3] = new SalariedEmployee("Coralie", "S", "Miller", Person.Sex.FEMALE,
17                                         new DateTime(1967,11,21), "44444444", 67000.00);
18
19       for(int i=0; i<employees.Length; i++){
20         Console.WriteLine(employees[i].FullName + " " + String.Format("{0:C}", employees[i].Pay()));
21       }
22     } // end Main()
23   } // end class definition
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 11.20 — on line 5, the EmployeeTestApp program declares an array of Employee references named employees. On lines 7 through 17, it initializes each Employee reference to point to either an HourlyEmployee or SalariedEmployee object.

In the `for` statement on line 19, each Employee object is manipulated polymorphically via the interface specified by the Employee class, which includes the interfaces inherited from Person and IPayable. The results of running this program are shown in Figure 11-19.

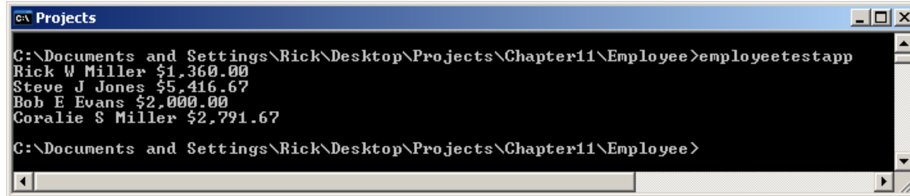


Figure 11-19: Results of Running Example 11.20

Inheritance Example: Engine Simulation

This example expands on the engine simulation originally presented in Chapter 10. Here the concepts of inheritance fuse with compositional design to yield a truly powerful combination.

Engine Simulation UML Diagram

Figure 11-20 shows the UML diagram for this version of the engine simulation. Note now that most of the functionality of a part resides in the Part class. In addition to its constructor method, the Part class contains two private fields: _partStatus and _partName, and two public read-write properties: Status, PartName. It contains one read-only property named IsWorking, which simply returns true or false depending on the part's current status.

The IManagedPart interface declares two methods: SetFault() and ClearFault(). The EnginePart class extends the Part class and implements IManagedPart. The EnginePart class also contains one private field named _registeredEngineNumber and a corresponding public property named RegisteredEngineNumber. It has one read-only property named PartIdentifier that returns a string containing the name of the part and its registered engine number. It also defines a private utility method named DisplayStatus() that is called internally by the SetFault() and ClearFault() methods. The EnginePart class is declared to be an abstract class to prevent the creation of EnginePart objects with the `new` operator.

The classes OilPump, FuelPump, Compressor, WaterPump, OxygenSensor, and TemperatureSensor all extend from EnginePart. The Engine class is an aggregate of all of its parts. It contains an array of EngineParts named _itsParts. It also contains several other private fields, _engineNumber and _isRunning, along with their corresponding public properties EngineNumber and IsRunning. It has four public methods: StartEngine(), StopEngine(), SetPartFault(), and ClearPartFault(). It has one private method named CheckEngine(), which is used internally by the StartEngine() method.

Simulation Operational Description

Examples 11.31 and 11.32 give the source code for the Engine and EngineTestApp classes. Refer to them for this discussion. The results of running Example 11.32 can be seen in Figure 11-21.

Referring first to Example 11.32 — the EngineTestApp declares an Engine reference named e1 and creates an Engine object with an engine number of 1. This is followed by a call to both the StartEngine() and StopEngine() methods. Next, a fault is set in the OilPump via a call to the SetPartFault() method. An attempt is then made to call StartEngine(), but because of the now faulty oil pump the engine will not start. The fault is cleared with a call to ClearPartFault(), and the next call to StartEngine() works fine. You can follow this sequence of events in Figure 11-21.

 © 2008 Rick Miller — All Rights Reserved

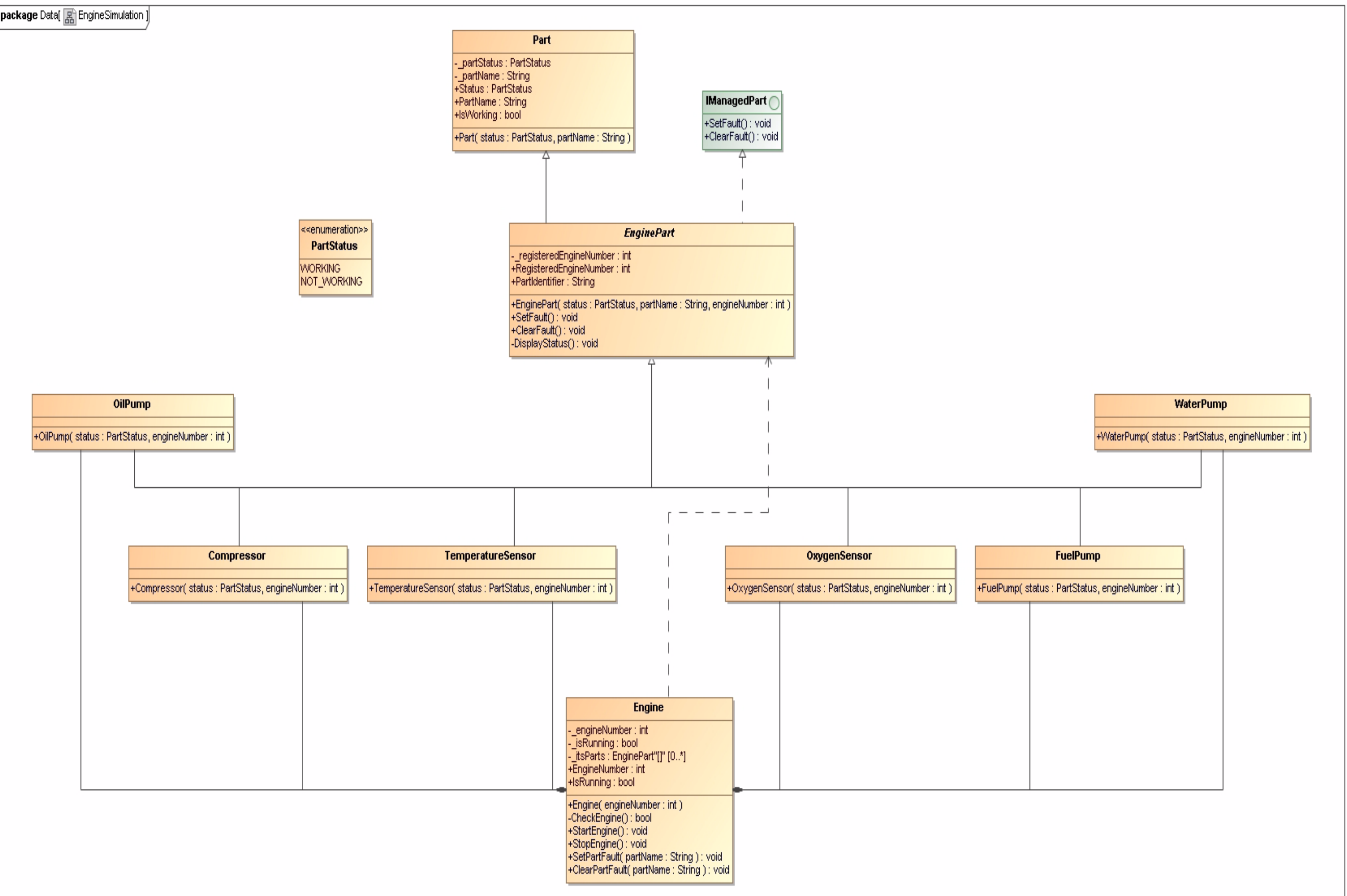


Figure 11-20: Engine Simulation UML Class Diagram

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\EngineSimulation>enginetestapp
Part Created...
Compressor created...
Part Created...
FuelPump created...
Part Created...
OilPump created...
Part Created...
WaterPump created...
Part Created...
OxygenSensor created...
Part Created...
TemperatureSensor created...
Engine number 1 created
Checking engine number 1...
Engine number 1 working properly!
Engine number 1 started!
Engine number 1 has been stopped!
OilPump For Engine Number: 1 status is now: NOT_WORKING
The status of Engine number 1's OilPump is NOT_WORKING
Checking engine number 1...
OilPump For Engine Number: 1 NOT_WORKING
Engine number 1 malfunction!
Engine number 1 is not running!
Engine number 1 failed to start!
OilPump For Engine Number: 1 status is now: WORKING
The status of Engine number 1's OilPump is WORKING
Checking engine number 1...
Engine number 1 working properly!
Engine number 1 started!
Engine number 1 has been stopped!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\EngineSimulation>
```

Figure 11-21: Results of Running the EngineTestApp

Compiling The Engine Simulation Code

You can compile the engine simulation code by putting all the code in one directory and issuing the following command: `csc *.cs`

Complete Engine Simulation Code Listing

11.21 PartStatus.cs

```
1    using System;
2
3    namespace EngineSimulation {
4
5      public enum PartStatus { WORKING, NOT_WORKING }
6
7    }
```

11.22 Part.cs

```
1    using System;
2
3    namespace EngineSimulation {
4
5      public class Part {
6        private PartStatus _partStatus;
7        private String _partName;
8
9        public PartStatus Status {
10         get { return _partStatus; }
11         set { _partStatus = value; }
12       }
13
14       public String PartName {
15         get { return _partName; }
16         set { _partName = value; }
17       }
18
19       public bool IsWorking {
20         get {
21           if (Status == PartStatus.WORKING) {
22             return true;
23           }
24           return false;
```

 © 2008 Rick Miller — All Rights Reserved

```
25          }
26        }
27
28        public Part(PartStatus status, String partName) {
29          PartName = partName;
30          Status = status;
31          Console.WriteLine("Part Created...");
32        }
33      } // end class definition
34    } // end namespace
```

11.23 IManagedPart.cs

```
1     namespace EngineSimulation {
2       public interface IManagedPart {
3         void SetFault();
4         void ClearFault();
5       }
6     }
```

11.24 EnginePart.cs

```
1     using System;
2
3     namespace EngineSimulation {
4       public abstract class EnginePart : Part, IManagedPart {
5        private int _registeredEngineNumber;
6
7        public int RegisteredEngineNumber {
8          get { return _registeredEngineNumber; }
9          set { _registeredEngineNumber = value; }
10       }
11
12       public String PartIdentifier {
13         get { return PartName + " For Engine Number: " + RegisteredEngineNumber; }
14       }
15
16       public EnginePart(PartStatus status, String partName, int engineNumber ):base(status, partName){
17         RegisteredEngineNumber = engineNumber;
18       }
19
20       public void SetFault() {
21         Status = PartStatus.NOT_WORKING;
22         DisplayStatus();
23       }
24
25       public void ClearFault() {
26         Status = PartStatus.WORKING;
27         DisplayStatus();
28       }
29
30       private void DisplayStatus(){
31         Console.WriteLine(PartIdentifier + " status is now: " + Status);
32       }
33      } // end class definition
34    } // end namespace
```

11.25 Compressor.cs

```
1     using System;
2
3     namespace EngineSimulation {
4       public class Compressor : EnginePart {
5        public Compressor(PartStatus status, int engineNumber):
6                  base(PartStatus.WORKING, "Compressor", engineNumber){
7          Console.WriteLine("Compressor created...");
8        }
9       } // end class
10    } // end namespace
```

11.26 FuelPump.cs

```
1     using System;
2
3     namespace EngineSimulation {
4       public class FuelPump : EnginePart {
5        public FuelPump(PartStatus status, int engineNumber):
6                   base(PartStatus.WORKING, "FuelPump", engineNumber){
7            Console.WriteLine("FuelPump created...");
8          }
9       } // end class
10    } // end namespace
```


© 2008 Rick Miller — All Rights Reserved

11.27 OilPump.cs

```
1    using System;
2
3    namespace EngineSimulation {
4      public class OilPump : EnginePart {
5       public OilPump(PartStatus status, int engineNumber):
6                base(PartStatus.WORKING, "OilPump", engineNumber){
7          Console.WriteLine("OilPump created...");
8       }
9      } // end class
10   } // end namespace
```

11.28 OxygenSensor.cs

```
1    using System;
2
3    namespace EngineSimulation {
4      public class OxygenSensor : EnginePart {
5       public OxygenSensor(PartStatus status, int engineNumber):
6                          base(PartStatus.WORKING, "OxygenSensor", engineNumber){
7          Console.WriteLine("OxygenSensor created...");
8       }
9      } // end class
10   } // end namespace
```

11.29 TemperatureSensor.cs

```
1    using System;
2
3    namespace EngineSimulation {
4      public class TemperatureSensor : EnginePart {
5       public TemperatureSensor(PartStatus status, int engineNumber):
6                       base(PartStatus.WORKING, "TemperatureSensor", engineNumber){
7          Console.WriteLine("TemperatureSensor created...");
8       }
9      } // end class
10   } // end namespace
```

11.30 WaterPump.cs

```
1    using System;
2
3    namespace EngineSimulation {
4      public class WaterPump : EnginePart {
5       public WaterPump(PartStatus status, int engineNumber):
6                base(PartStatus.WORKING, "WaterPump", engineNumber){
7          Console.WriteLine("WaterPump created...");
8       }
9      } // end class
10   } // end namespace
```

11.31 Engine.cs

```
1    using System;
2
3    namespace EngineSimulation {
4       public class Engine {
5          private int _engineNumber;
6          private bool _isRunning;
7          private EnginePart[] _itsParts;
8
9          public int EngineNumber {
10            get { return _engineNumber; }
11            set { _engineNumber = value; }
12         }
13
14         public bool IsRunning {
15            get { return _isRunning; }
16            set { _isRunning = value; }
17         }
18
19         public Engine(int engineNumber){
20            EngineNumber = engineNumber;
21            IsRunning = false;
22            _itsParts = new EnginePart[6];
23            _itsParts[0] = new Compressor(PartStatus.WORKING, EngineNumber);
24            _itsParts[1] = new FuelPump(PartStatus.WORKING, EngineNumber);
25            _itsParts[2] = new OilPump(PartStatus.WORKING, EngineNumber);
26            _itsParts[3] = new WaterPump(PartStatus.WORKING, EngineNumber);
27            _itsParts[4] = new OxygenSensor(PartStatus.WORKING, EngineNumber);
28            _itsParts[5] = new TemperatureSensor(PartStatus.WORKING, EngineNumber);
```

 © 2008 Rick Miller — All Rights Reserved

```
29           Console.WriteLine("Engine number {0} created", EngineNumber);
30        }
31
32
33        private bool CheckEngine(){
34           Console.WriteLine("Checking engine number {0}...", EngineNumber);
35           bool is_working = false;
36
37           for(int i=0; i<_itsParts.Length; i++){
38              is_working = _itsParts[i].IsWorking;
39              if(!is_working){
40                 Console.WriteLine(_itsParts[i].PartIdentifier + " " + _itsParts[i].Status);
41                 break;
42
43                 }
44           }
45
46           if(is_working){
47              Console.WriteLine("Engine number {0} working properly!", EngineNumber);
48           }else{
49              Console.WriteLine("Engine number {0} malfunction!", EngineNumber);
50              StopEngine();
51           }
52           return is_working;
53        }
54
55
56
57        public void StartEngine(){
58           if(!IsRunning){
59              IsRunning = CheckEngine();
60              if(!IsRunning){
61                 Console.WriteLine("Engine number {0} failed to start!", EngineNumber);
62              }else{
63                 Console.WriteLine("Engine number {0} started!", EngineNumber);
64              }
65           }else{
66              Console.WriteLine("Engine number {0} is already running!", EngineNumber);
67           }
68        }
69
70        public void StopEngine(){
71           if(IsRunning){
72              IsRunning = false;
73              Console.WriteLine("Engine number {0} has been stopped!", EngineNumber);
74           }else{
75              Console.WriteLine("Engine number {0} is not running!", EngineNumber);
76           }
77        }
78
79
80        public void SetPartFault(String partName){
81           for(int i=0; i<_itsParts.Length; i++){
82              if(_itsParts[i].PartName.Equals(partName)){
83                 _itsParts[i].SetFault();
84                 Console.WriteLine("The status of Engine number {0}'s {1} is {2}", EngineNumber,
85                                   _itsParts[i].PartName, _itsParts[i].Status);
86              }
87           }
88        }
89
90        public void ClearPartFault(String partName){
91           for(int i=0; i<_itsParts.Length; i++){
92              if(_itsParts[i].PartName.Equals(partName)){
93                 _itsParts[i].ClearFault();
94                 Console.WriteLine("The status of Engine number {0}'s {1} is {2}", EngineNumber,
95                                         _itsParts[i].PartName, _itsParts[i].Status);
96              }
97           }
98        }
99
100     } // end class
101  } // end namespace
```

 © 2008 Rick Miller — All Rights Reserved

11.32 EngineTestApp.cs

```
1    using System;
2    using EngineSimulation;
3
4    public class EngineTestApp {
5       public static void Main(){
6          Engine e1 = new Engine(1);
7          e1.StartEngine();
8          e1.StopEngine();
9          e1.SetPartFault("OilPump");
10         e1.StartEngine();
11         e1.ClearPartFault("OilPump");
12         e1.StartEngine();
13         e1.StopEngine();
14      } // end Main()
15   } // end class
```

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 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.

Class and interface constructs introduce new, user-defined data types. The interface construct is used to specify a set of authorized type operations but omits their behavior; the class construct is used to specify a set of authorized type operations and, optionally, their behavior as well. A class construct, like an interface, can omit the bodies of one or more of its members, however, such members must be declared to be abstract. A class that declares one or more of its members 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 public, protected, internal, and protected internal members 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* as 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, use casting 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 overriding. Use the `virtual` keyword to declare an overrideable base class method. Use the `override` keyword to define an overriding derived class method. Overriding methods can be called polymorphically via a base class reference that points to a derived class object.

An *abstract member* is a member that omits its body and has no implementation behavior. A class that declares one or more abstract members must be declared to be abstract. The primary purpose of an *abstract class* is to provide a specification for behavior whose implementation is 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 behavior. Interfaces can have four types of members: 1) properties, 2) methods, 3) events, and 4) indexers. 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.

Use the access modifiers `private`, `protected`, `public`, `internal`, and `protected internal` to control horizontal and vertical member access.

Use the `sealed` keyword to stop the inheritance mechanism or prevent base class methods from being overridden in derived classes.

© 2008 Rick Miller — All Rights Reserved

Polymorphic behavior is achieved in a program by targeting a set of operations specified by a base class or interface and manipulating their derived class objects via those operations. 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. **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 console. Create a default constructor for ClassA that prints a message to the console 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. Call the methods A(), B(), and C() via each of the references. Run the test driver program and note the results.

2. **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 console. 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. Call the methods A(), B(), and C() via each of the references. Run the test driver program and note the results.

3. **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 console. 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. Call the methods A(), B(), and C() via each of the references. Run the program and note the results.

4. **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.

5. **Mental Exercise:** Consider the following scenario: 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.

   Answer these questions: 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 Engine constructor call. Refer to the code supplied in Examples 11.21 through 11.32.

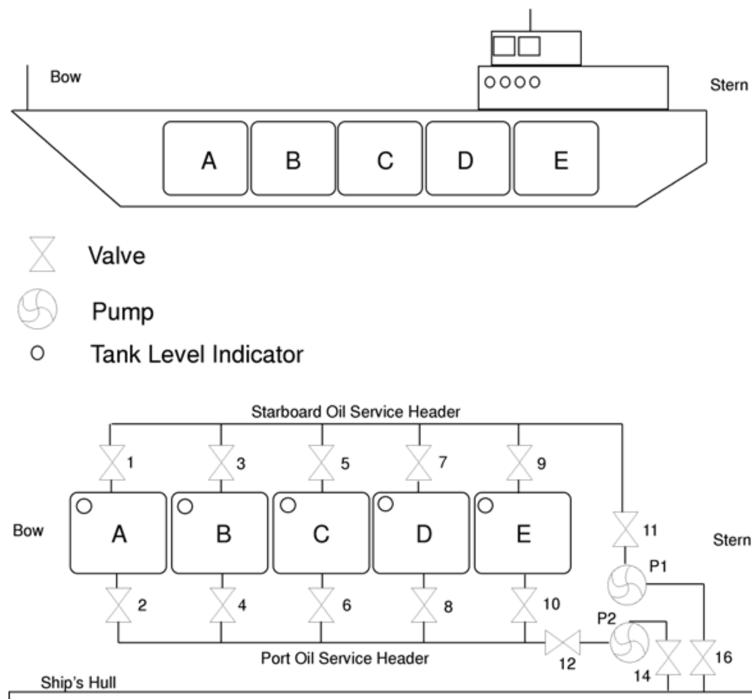
© 2008 Rick Miller — All Rights Reserved

2. **Draw Sequence Diagram:** Draw a UML sequence diagram of the Engine StartEngine() method.

3. **Draw Sequence Diagram:** Draw a UML sequence diagram of the Engine CheckEngine() method.

4. **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.

5. **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. **Note:** 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 1,000 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 and 16, and start pumps P1 and P2 in the fill direction in the fast mode.

© 2008 Rick Miller — All Rights Reserved

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 those methods. What type of class am I?

12. List the four authorized members of an interface.

13. Which 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 of using the access modifiers `public`, `private`, `protected`, `internal`, and `protected internal` has on horizontal and vertical member access.

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

© 2008 Rick Miller — All Rights Reserved

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 OOPSLA '00 Conference Proceedings.

Derek Rayside and Gerard T. Campbell. *An Aristotelian Understanding of Object-Oriented Programming*. OOPSLA '00 Conference Proceedings.

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Rick Miller. *Java For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-05-2

Notes

© 2008 Rick Miller — All Rights Reserved

Part III: Graphical User Interface Programming
&
Custom Events

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Chapter 12

Pentax 67 / SMC Takumar 55/2.8 / Kodak Tri-X Professional

Fairview Park

Windows Forms Programming

Learning Objectives

- *Create Graphical User Interface (GUI) programs using Windows forms components*
- *List and describe the parts of a window*
- *Describe how to register event handler methods with component events*
- *State the definition of the term "delegate"*
- *Handle events generated in one object using event handler methods located in another object*
- *Use the Form, TextBox, Button, and Label controls*
- *Automatically arrange controls with the FlowLayoutPanel and TableLayoutPanel controls*
- *Pass references to event handler objects via constructor methods*
- *Manipulate arrays of controls*
- *Add controls to a window's Controls collection*
- *Control programs via menus*
- *Manipulate text in a text box*

© 2008 Rick Miller — All Rights Reserved

Introduction

Nearly every application running on your personal computer (PC) sports a Graphical User Interface (GUI). This chapter shows you how to create GUIs for your programs.

Before we get started, I'd like to share with you some good news and some bad news. First the bad news: An exhaustive treatment of all aspects of Microsoft Windows GUI programming is way beyond the scope of this book. If you want to move beyond what's covered in this chapter, I recommend reading one of the many books available devoted entirely to the subject. Go to any good book store and you'll find several on the shelves.

Now the good news: You don't need to know a whole lot to create really nice GUIs. Most of the heavy lifting is done for you by the classes found in the System.Windows.Forms namespace. Some of the more important classes to know include *Form*, *TextBox*, *Button*, and *Label*. Add to these an understanding of how events and delegates work and you'll be off to a good start.

I will also teach you how to separate the GUI from other parts of your program. To do this, you'll need to know how to register event handler methods, located in one or more separate classes, with buttons or other GUI components located in your GUI.

An unfortunate mistake many novice programmers make is to rely too heavily upon the GUI designer available in Microsoft Visual Studio. The problem with using the GUI designer is that it makes it difficult to separate concerns. In this chapter, I will show you how to create GUIs by hand. It's not difficult once you get the hang of things. I'll also show you how you can automatically place components within a GUI via layout managers.

After you complete this chapter, the only thing you'll need to do to create spectacular GUIs is to dive deeper into the .NET API and use a little imagination.

The Form Class

The Form class, found in the System.Windows.Forms namespace, serves as the basis for all types of windows you might need to create in your application. These include *standard*, *tool*, *borderless*, or *floating* windows. The Form class is also used to create *dialog boxes* and *multiple-document interface* (MDI) windows.

Form Class Inheritance Hierarchy

A Form is a lot of things, as you can see from its inheritance diagram shown in Figure 12-1.

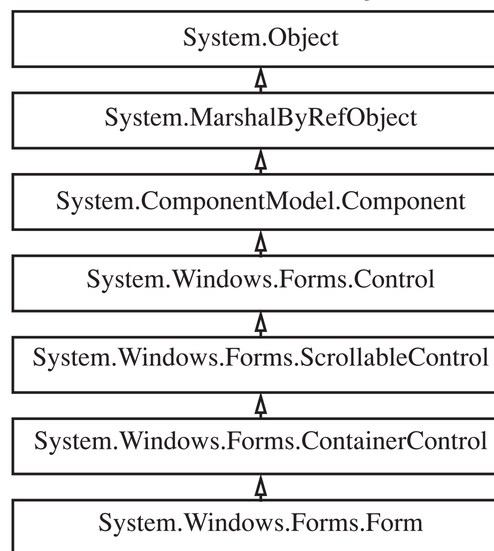


Figure 12-1: Form Class Inheritance Hierarchy

© 2008 Rick Miller — All Rights Reserved

Referring to Figure 12-1 — a Form is a *ContainerControl*, a *ScrollableControl*, a *Control*, a *Component*, a *MarshalByRefObject*, and ultimately an *Object*. I recommend that you visit the Microsoft Developer Network (MSDN) website and do a little research on the Form class so you can get a better feel for what it can do. A quick review of its methods and properties will give you a few ideas about how you can manipulate the Form class in your programs.

A Simple Form Program

Example 12.1 gives the code for a simple Form-based program. All this program does is display an empty window on the screen. I'll use its output to introduce you to the parts of a standard window.

12.1 SimpleForm.cs

```
1    using System;
2    using System.Windows.Forms;
3
4    public class SimpleForm : Form {
5       public static void Main(){
6          Application.Run(new SimpleForm());
7       }
8    }
```

Referring to Example 12.1 — the SimpleForm class extends Form and provides a Main() method. The important point to note here is the use on line 6 of the System.Windows.Forms.Application class to display the form. The Application class's static Run() method starts an *application message loop* on the current thread. Don't worry about threads for now as they are covered in detail in Chapter 16. I will discuss messages and the message loop in more detail in the next section.

A Microsoft Windows program is event-driven, meaning that when a window is displayed, it will sit there forever processing events until the application exits. Some of the events are mouse clicks within the window itself or on controls within the window like buttons, text boxes, or menus. Other events may be events sent to the application from other applications, like the operating system, perhaps.

You can compile Example 12.1 two ways. If you compile it the way we've been compiling programs up untill now, which is just using `csc *.cs`, you will create an application that displays both a window and a command console. Compiling with the `/target:winexe` switch results in an application that displays only the window. The full command required to create a windows executable from Example 12.1 is:

```
csc /target:winexe SimpleForm.cs
```

You can compile either way, but you'll find having a console window to display output can come in handy for testing purposes, as you'll see later. Figure 12-2 shows the results of running Example 12.1.

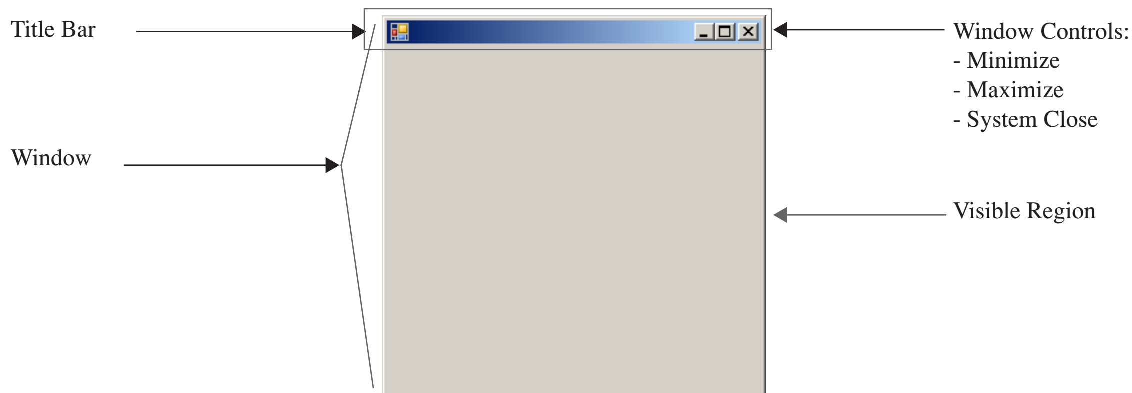


Figure 12-2: Results of Running Example 12.1

Referring to Figure 12-2 — when you execute the program either from the command line or by double-clicking, it displays an empty window. It's only empty because I didn't put anything in it, nor did I set any of its properties. However, the empty window has a lot of functionality built in. You can drag the window around the screen, resize the window, minimize the window, maximize the window, and close the application by clicking the box with the "X" in the upper right corner. It has all the basic functionality you've come to expect from a standard window.

The title bar would have a title in it if I had set the form's *Text* property. I'll show you how to do that in a moment. The window's visible region includes those areas of the window visible to the user. In this case the entire

© 2008 Rick Miller — All Rights Reserved

window is visible. If you moved another window over top of this one, then some of it would be visible and some of it would not. Figure 12-3 shows the same window resized smaller and larger.

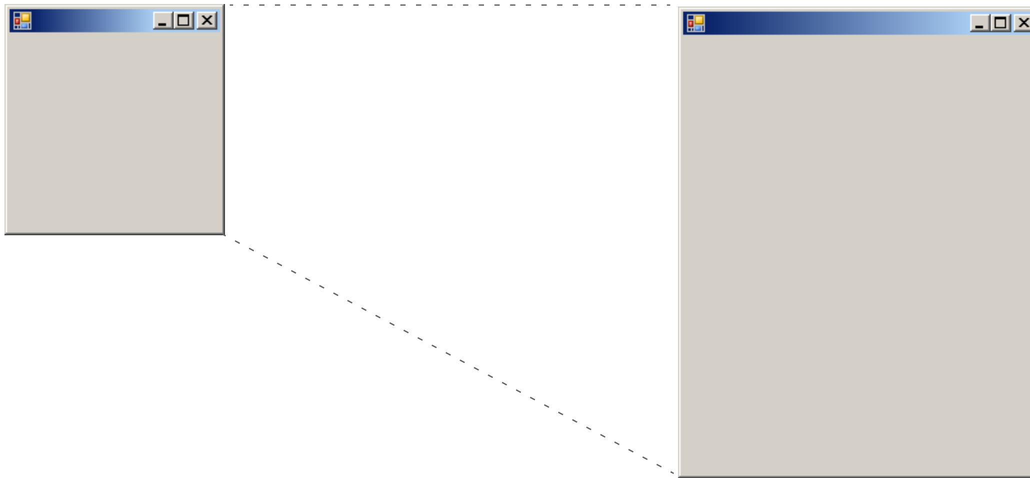


Figure 12-3: A Standard Window can be Resized by Dragging the Lower Right Corner

Quick Review

The Form class, found in the System.Windows.Forms namespace, serves as the basis for all types of windows you might need to create in your application. These include *standard*, *tool*, *borderless*, or *floating* windows. The Form class is also used to create *dialog boxes* and *multiple-document interface* (MDI) windows. A Form is a ContainerControl, a ScrollableControl, a Control, a Component, a MarshalByRefObject, and ultimately an Object.

The Form class provides a lot of functionality right out of the box. You can drag the window around the screen, resize the window, minimize the window, maximize the window, and close the application by clicking the box with the "X" in the upper right corner.

Application Messages, Message Pump, Events, And Event Loop

As I mentioned earlier, Microsoft Windows applications are *event driven*. This means that when a GUI application executes, it sits there patiently waiting for an event to occur such as a mouse click or keystroke. These events are delivered to the application in the form of *messages*. Messages can be generated by the system in response to various types of stimuli including direct user interaction (*i.e.*, mouse movement, clicking, scrolling, keystrokes, etc.), or by other applications.

The majority of system-generated messages are placed into a data structure referred to as the *system message queue*. A queue is a data structure that has a first-in-first-out (FIFO) characteristic. Messages are placed into the queue as they occur. Each message contains data that includes the type of event, target window, cursor coordinates, etc. The system then examines each message and routes it to its target window. This process, referred to as the *message pump*, is illustrated in Figure 12-4.

Referring to Figure 12-4 — system messages are placed into the system message queue where they wait in line to be processed. The system then examines the data within each message to determine its GUI application target. Each GUI application, which runs in its own thread of execution, has its own message queue. The message is placed in the GUI application's queue where again it waits its turn to be examined and forwarded on to its generating window. **Note:** Applications can have multiple windows open at the same time.

© 2008 Rick Miller — All Rights Reserved

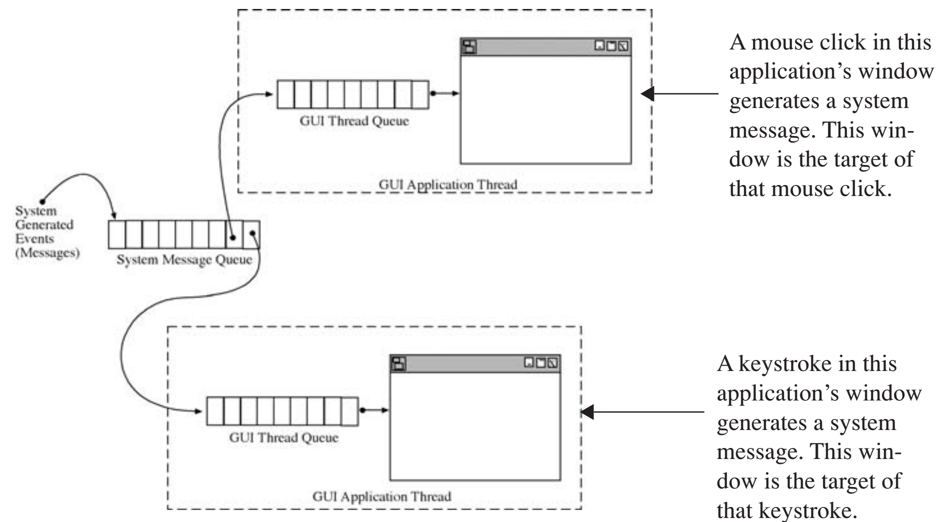


Figure 12-4: Windows Message Routing (Message Pump)

Message Categories

As you can well imagine, many types of events can occur during the execution of a complex GUI application. Each of these application events generates a corresponding system message. The following table lists the system message categories and their message prefixes.

Prefix	Message Category	Prefix	Message Category
ABM	Application Desktop Toolbar	**MCM**	Month Calendar Control
BM	Button Control	**PBM**	Progress Bar
CB	Combo Box	**PGM**	Pager Control
CBEM	Extended Combo Box Control	**PSM**	Property Sheet
CDM	Common Dialog Box	**RB**	Rebar Control
DBT	Device	**SB**	Status Bar Window
DL	Drag List Box	**SBM**	Scroll Bar Control
DM	Default Push Button Control	**STM**	Static Control
DTM	Date and Time Picker Control	**TB**	Toolbar
EM	Edit Control	**TBM**	Trackbar
HDM	Header Control	**TCM**	Tab Control
HKM	Hot Key Control	**TTM**	Tooltip Control
IPM	IP Address Control	**TVM**	Tree-view Control
LB	List Box Control	**UDM**	Up-down Control
LVM	List View Control	**WM**	General Window

Table 12-1: System Message Categories and their Prefixes

© 2008 Rick Miller — All Rights Reserved

Messages In Action: Trapping Messages With IMessageFilter

One way to see messages in action is to print them to the console as they occur. The following program is very similar to the SimpleForm code given in Example 12.1 in that the MessagePumpDemo class extends Form. It also implements the *IMessageFilter* interface, which declares one method named *PreFilterMessage()*. The PreFilterMessage() method's implementation begins on line 6. All it does in this simple example is write the incoming message to the console.

12.2 MessagePumpDemo.cs

```
1     using System;
2     using System.Windows.Forms;
3
4     public class MessagePumpDemo : Form, IMessageFilter {
5
6        public bool PreFilterMessage(ref Message m){
7           Console.WriteLine(m);
8           return false;
9        }
10
11       public static void Main(){
12          MessagePumpDemo mpd = new MessagePumpDemo();
13          Application.AddMessageFilter(mpd);
14          Application.Run(mpd);
15       }
16    }
```

Referring to Example 12.2 — any class that implements IMessageFilter can be used as a message filter. In this example, the MessagePumpDemo class uses an instance of itself as a message filter with a call to the Application.AddMessageFilter() method. To see the messages being printed to the console, compile this program into an ordinary console executable file. Figure 12-5 shows the results of running this program.

Figure 12-5: Results of Running Example 12.2

Referring to Figure 12-5 — cursor movement within the application window, not the console, causes the generation of WM_MOUSEMOVE messages. Note that since the window has no additional components like buttons or text boxes, almost all the messages generated belong to the WM (general window) category. Scrolling the mouse wheel causes a WM_MOUSEWHEEL message. Keystrokes cause a sequence of messages including WM_KEYDOWN, WM_CHAR, and WM_KEYUP. The best way to see these messages in action is to run this application and experiment with different types of mouse and keyboard entry along with window movement and resizing.

Final Thoughts On Messages

The information presented in this section falls into the category of "nice to know". Unless you're writing complex GUI applications that need to filter system messages you can safely ignore them. What you'll most likely do is create windows that contain various components, like text boxes, buttons, labels, menus, etc. These components, and indeed, forms as well, can respond to certain events. For example, buttons have (among others) the Click event. If

 © 2008 Rick Miller — All Rights Reserved

you want a button to do something in response to a mouse click on it, you will need to create what is referred to as an event handler method and register it with the button's Click event. When the button is clicked, its event handler method, or methods if there is more than one, is called.

So, although the system is creating, sending, and responding to messages, you will think in terms of components and the events they can respond to. I will show you how to do this shortly, but first, I want to show you a few things about screen coordinates.

Quick Review

Microsoft Windows applications are *event-driven*. When launched, they wait patiently for an event to occur such as a mouse click or keystroke. Events are delivered to the application in the form of *messages*. Messages can be generated by the operating system in response to various types of stimuli, including direct user interaction (i.e., mouse movement, clicking, scrolling, keystrokes, etc.), or by other applications.

The majority of system generated messages are placed into a data structure referred to as the *system message queue*. A queue is a data structure that has a FIFO characteristic. Messages are placed into the queue as they occur. Each message contains data that includes the type of event, target window, cursor coordinates, etc. The system then examines each message and routes it to its target window.

Screen And Window (Client) Coordinate System

When working with GUIs you'll need to be aware to two types of coordinates: *screen coordinates* and *window coordinates*. Window coordinates are also referred to as *client coordinates*.

A window is drawn upon a computer screen at a certain position. The placement of the window's upper left corner falls on a certain point within the screen's coordinate system. The basic unit of measure for a screen is the *pixel*. The screen coordinate system is illustrated in Figure 12-6.

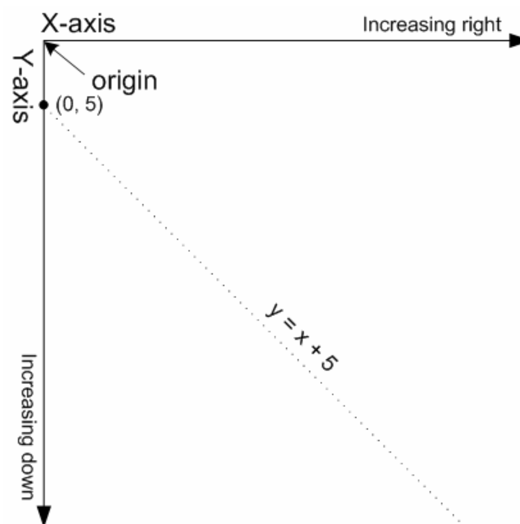


Figure 12-6: Screen Coordinate System

Referring to Figure 12-6 — the origin of the screen, or the point where the value of both its x and y coordinates equal 0, is located in the upper left corner of the screen. The value of the x coordinate increases to the right, while the value of the y coordinate increases down. Points upon the screen are given in (x, y) pairs.

Windows have a coordinate system similar to screen coordinates as is shown in Figure 12-7.

© 2008 Rick Miller — All Rights Reserved

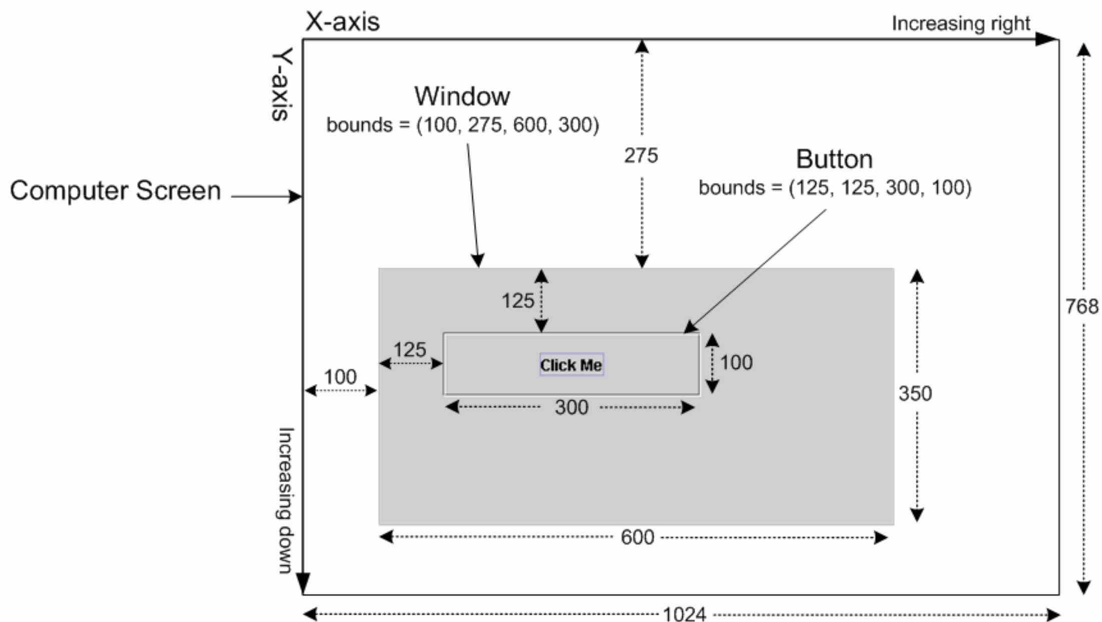


Figure 12-7: Window Coordinates

Referring to Figure 12-7 — the window is placed on the screen at position (100, 275). This is the location of its upper left corner. The *origin* of the window is its upper left corner, meaning that components drawn within the window are placed with respect to the window's origin. The button drawn in the window is placed at position (125, 125).

Windows, and the components drawn within them, have *height* and *width*. The bounds of a component are the location of its upper left corner together with its width and height. If a window is placed at position (100, 275) and is 300 pixels wide and 100 pixels high, then its bounds are (100, 275, 300, 100). Example 12.3 gives a short program that prints the bounds of a window in response to user input.

12.3 ShowBounds.cs

```
1    using System;
2    using System.Windows.Forms;
3
4    public class ShowBounds : Form, IMessageFilter {
5
6       public bool PreFilterMessage(ref Message m){
7          Console.WriteLine(this.Bounds);
8          return false;
9       }
10
11      public static void Main(){
12         ShowBounds sb = new ShowBounds();
13         Application.AddMessageFilter(sb);
14         Application.Run(sb);
15      }
16   }
```

Referring to Example 12.3 — this program is just a slight modification to the previous program. The ShowBounds class extends Form and implements the IMessageFilter interface. The PreFilterMessage() method has been modified to print the window's Bounds property. Figure 12-8 shows the results of running this program.

Referring to Figure 12-8 — the output shown is the result of dragging the window through various sizes. Its final screen position is (45, 136); its final width is 271 pixels wide, and it is 111 pixels high. Thus, the bounds of this particular window are (45, 136, 271, 111), as is shown in the console window's final lines of output.

 © 2008 Rick Miller — All Rights Reserved

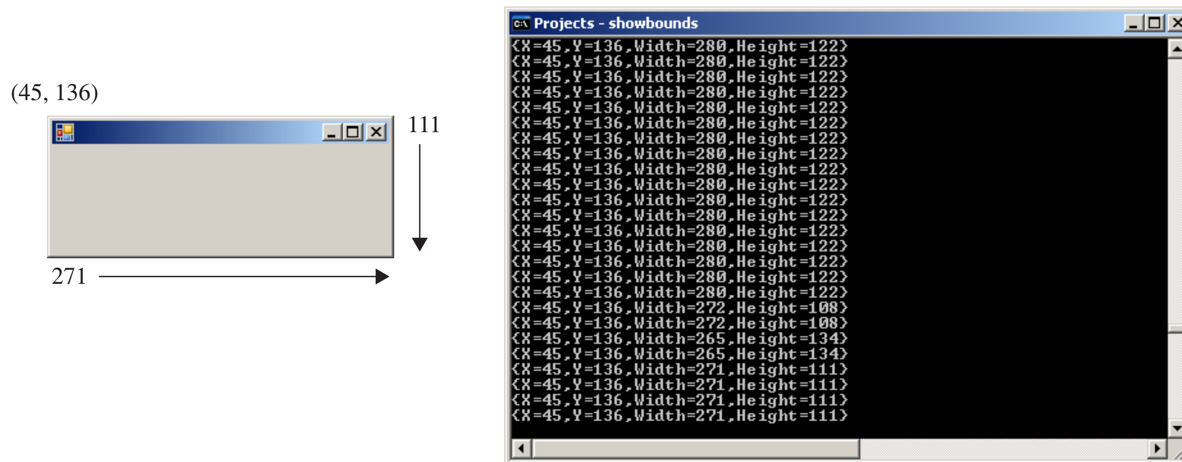


Figure 12-8: Results of Running Example 12.3

Quick Review

There are two types of coordinates: *screen coordinates* and *window coordinates*. Window coordinates are also referred to as *client coordinates*. The basic unit of measure upon a screen is the *pixel*. The origin of the screen, or the point where the value of both its x and y coordinates equal 0, is located in the upper left corner of the screen. The value of the x coordinate increases to the right, while the value of the y coordinate increases down. Points upon the screen are given in (x, y) pairs. Windows have a coordinate system similar to the screen, with their *origin* located in the upper left corner of the window. Windows, and the components drawn within them, have *height* and *width*. The *bounds* of a component are the location of its upper left corner together with its width and height.

Manipulating Form Properties

The Form class provides many properties, methods, and events that make it easy to manipulate them in your programs. In fact, as you saw in Figure 12-1, the Form class gets most of its functionality via its inheritance hierarchy. What you can do to a form you can also do, for the most part, to other controls, containers, and components.

In this section I'd like to demonstrate a few helpful form properties. Before going on though, I'd like to say that there are way too many Form class members to demonstrate them all in one section, or even one chapter. I recommend you take the time now, if you haven't already done so, to review the Form class documentation on the MSDN website and get a feel for all the things you can do to a form. I will demonstrate the use of other Form members when their use becomes appropriate in the text.

When you display a window, it's usually nice to give it a title. You can set a window's title bar text via its *Text* property. If you want to change a window's background color set its *BackColor* property. If you want to set a window's background image, do so via its *BackgroundImgage* property.

The use of each of these properties requires the help of additional .NET Framework classes or structures including System.Drawing.*Color*, System.Drawing.*Image*, and System.Drawing.*BitMap*. Example 12.4 gives a short program that shows how to set a window's title bar text, backgound color, and its background image.

12.4 FormProperties.cs

```
1     using System;
2     using System.Windows.Forms;
3     using System.Drawing;
4
5     public class FormPropertiesDemo : Form {
6       public static void Main(String[] args){
7         FormPropertiesDemo fpd = new FormPropertiesDemo();
8         if(args.Length > 0){
9           try{
10            Image image = new Bitmap(args[0]);
11            fpd.BackgroundImage = image;
```

 © 2008 Rick Miller — All Rights Reserved

```
12            fpd.Size = image.Size;
13          }catch(Exception){
14             //ignore for now
15          }
16        }else{
17           fpd.BackColor = Color.Black;
18        }
19
20        fpd.Text = "Form Properties Demo";
21        Application.Run(fpd);
22
23     } // end Main()
24   } // end class
```

Referring to Example 12.4 — the FormProperties class extends Form. Notice that I have used the String array version of the Main() method. This program can be run two ways: 1) by providing the name of an image file to use as the window background image, or 2) with no command line input, in which case the window's background color is set to black.

The code that creates the image and sets the window's BackgroundImage property is enclosed in a `try/catch` block that ignores the generated exception. If an exception does occur, the window is displayed with its default background color and no image. It would be easy, however, to add some code to the body of the *catch* clause that sets the background image to some default image.

Notice on line 11 that the image is created with the help of the Bitmap class. The Bitmap class has many overloaded constructor methods that make it easy to create images from different sources. The version used here creates an image from a string that represents the image filename. The string can be just the name of the file, in which case the program expects to find the image file in the default or working directory. (*i.e.*, The directory in which it executes.) The string can also be a complete path name. Notice on line 12 that the size of the window is set to the size of the image.

If the program is run with no command-line argument or by being double-clicked, then its background color is set to black with the help of the Color structure on line 17. The Color structure defines many public properties that represent different colors. In this example, I used Color.Black to set the window's BackColor property.

Lastly, I set the window's title bar text on line 20 via its Text property. I then display the form and kick off the GUI application execution thread with the Application.Run() method.

Figure 12-9 shows the results of running this program via the command line given the name of an image file. (You can download this image, WCC_2.jpg, from the PulpFreePress.com or Warrenworks.com websites, or you can use one of your own images.)

Figure 12-9: Running Example 12.4 via the Command Line with the Name of the Image WCC_2.jpg

 © 2008 Rick Miller — All Rights Reserved

Figure 12-10 shows the results of running Example 12.4 with no image name or by double-clicking the executable file.

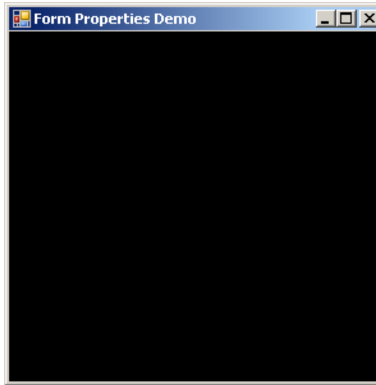


Figure 12-10: Running Example 12.4 with no Image

Quick Review

The Form class provides many properties, methods, and events which make it easy to manipulate them in your programs. The Form class gets most of its functionality via its inheritance hierarchy. What you can do to a form you can also do, for the most part, to other controls, containers, and components.

Setting a control's properties often requires the use of other classes, structures, or enumerations found in the .NET Framework. A few of these include System.Drawing.*Point*, System.Drawing.*Rectangle*, System.Drawing.*Color,* System.Drawing.*Bitmap*, and System.Drawing.*Image*. The type of property determines what type of object you must use to set the property.

Adding Components To Windows: Button, TextBox, And Label

In this section, I show you how to add components to windows. The components I use to explain the concepts include Button, TextBox, and Label. You'll find these, and many other components, in the System.Windows.Forms namespace. You'll also need the System.Drawing.*Point* structure to help place components in absolute positions within the window. Study the code given in Example 12.5.

12.5 ComponentDemo.cs

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4
5
6    public class ComponentDemo : Form {
7       private Button _button1;
8       private TextBox _textbox1;
9       private Label _label1;
10
11      public ComponentDemo(int x, int y, int width, int height){
12         this.Bounds = new Rectangle(x, y, width, height);
13         this.Text = "Component Demo";
14         InitializeComponents();
15      }
16
17      public ComponentDemo():this(100, 200, 400, 200){ }
18
19      private void InitializeComponents(){
20         _label1 = new Label();
21         _label1.Text = "This is a Label!";
22         _label1.Location = new Point(25, 25);
23
24         _button1 = new Button();
25         _button1.Text = "Click Me!";
```

 © 2008 Rick Miller — All Rights Reserved

```
26         _button1.Location = new Point(125, 25);
27
28         _textbox1 = new TextBox();
29         _textbox1.Text = "some default text";
30         _textbox1.Location = new Point(225, 25);
31
32         this.Controls.Add(_label1);
33         this.Controls.Add(_button1);
34         this.Controls.Add(_textbox1);
35       }
36
37       public static void Main(){
38         Application.Run(new ComponentDemo());
39       } // end Main()
40     } // end class
```

Referring to Example 12.5 — the ComponentDemo class extends Form and declares three private component members: *_button1*, *_textbox1*, and *_label1*. It declares two constructors. The first constructor, beginning on line 11, declares four parameters that are used to set the window's Bounds property. Notice that the Bounds property must be set with the help of a *Rectangle* structure. Alternatively, you could set the window's Location, Height, and Width properties separately. On line 13, the window's title bar text is set via its Text property. And lastly, on line 14, the InitializeComponents() method is called.

The InitializeComponents() method begins on line 19 and creates and initializes the window's _button1, _textbox1, and _label1 components. Notice how each component's Location property is set with the help of the Point structure.

On lines 32 through 34, each component is added to the window by adding it to the window's *Controls* collection. Figure 12-11 shows the results of running this program.

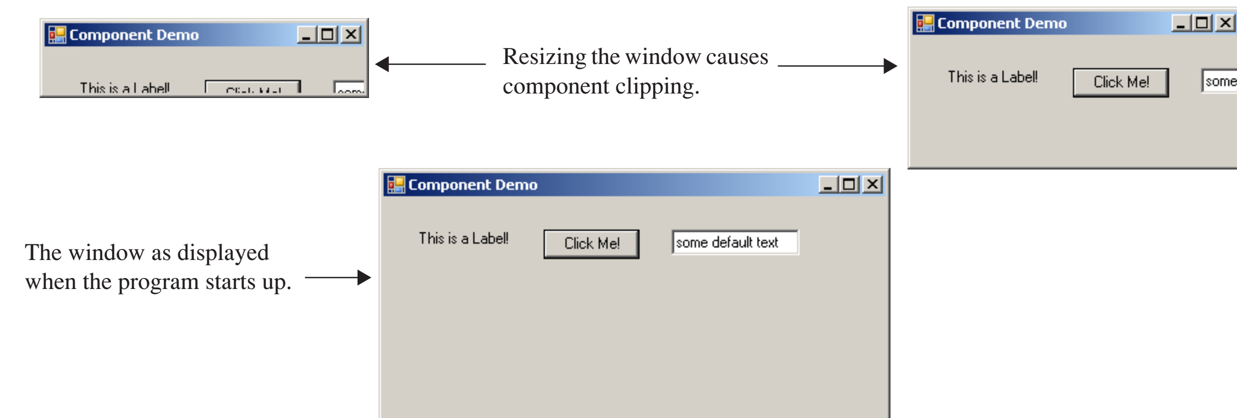


Figure 12-11: Results of Running Example 12.5

Referring to Figure 12-11 — notice the effects of setting the Text property for each component. Also note how the absolute placement of the components affects the window's appearance when it's resized. It's tedious to place components in specific positions within a window. If you're not careful, you can cover one component with another and wonder where it disappeared to. In a moment, I'll show you how to use layout panels to automatically place components within a window. But first, I want to show you how to make the button actually do something when it's clicked.

Quick Review

To add a control like a Button or TextBox to a window, you must first declare and create the control, set its properties, and then add the control to the window's Controls collection. The absolute placement of controls can be tedious. Use the System.Drawing.Rectangle structure to set a control's Bounds property. You may alternatively set a control's Top, Left, Width, and Height properties separately.

 © 2008 Rick Miller — All Rights Reserved

Registering Event Handlers With GUI Components

The whole point of creating a GUI is to have it respond to user interaction. As it stands now, the program shown in Example 12.5 simply displays a window with a label, a button, and a text box. Although you can type in the text box and click the button, nothing else happens. Let's change that by adding an event handler method and registering it with the button's Click event.

Delegates And Events

All System.Windows.Forms GUI controls have event members. An *event* is something an object can respond to. For example, a Button can respond to a mouse click via its Click event (and also via its MouseClick event!) GUI components can respond to many types of events. Table 12-2 offers an incomplete listing of some common events associated with the Control class, from which most Windows forms components inherit.

Event	Description	Delegate Type
BackColorChanged	Occurs when the BackColor property changes	System.EventHandler
BackgroundImageChanged	Occurs when the BackgroundImage property changes	System.EventHandler
Click	Occurs when control is clicked.†	System.EventHandler
DoubleClick	Occurs when control is double clicked	System.EventHandler
GotFocus	Occurs when the control receives focus	System.EventHandler
MouseClick	Occurs when the control is clicked by the mouse††	System.Windows.Forms.MouseEventHandler
MouseDoubleClick	Occurs when the control is double-clicked by the mouse	System.Windows.Forms.MouseEventHandler
MouseDown	Occurs when the mouse pointer is over the control and the mouse button is pressed	System.Windows.Forms.MouseEventHandler
MouseEnter	Occurs when the mouse pointer enters the control	System.EventHandler
MouseLeave	Occurs when the mouse pointer leaves the control	System.EventHandler
MouseMove	Occurs when the mouse pointer moves over the control	System.Windows.Forms.MouseEventHandler
MouseUp	Occurs when the mouse pointer is over the control and the mouse button is released	System.Windows.Forms.MouseEventHandler
Paint	Occurs when the control is redrawn	System.Windows.Forms.PaintEventHandler
† A Click event can be caused by pressing the Enter key †† A MouseClick is of type MouseEventHandler which uses MouseEventArgs to convey additional mouse information to the event handler method		

Table 12-2: Partial Listing of Control Events

The important thing to note about the events listed in Table 12-2 is that different types of events are handled by different types of event handlers. An event handler type is defined or specified by a *delegate* type. A delegate provides

© 2008 Rick Miller — All Rights Reserved

a specification for a method signature. For example, the System.EventHandler delegate specifies a method with the following signature:

```
void EventHandler(Object sender, EventArgs e)
```

This means that if you want to register a method to respond to Click events on a control, the method must have the same signature as the event's delegate type. The best way to see all this work is to look at some code. Example 12.6 expands on the previous example by adding an event handler method for the _button1 component. When the button is clicked the text appearing in the text box is used to set the label's text.

12.6 ComponentDemo.cs (Mod 1)

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4
5
6    public class ComponentDemo : Form {
7       private Button _button1;
8       private TextBox _textbox1;
9       private Label _label1;
10
11      public ComponentDemo(int x, int y, int width, int height){
12         this.Bounds = new Rectangle(x, y, width, height);
13         this.Text = "Component Demo";
14         InitializeComponents();
15      }
16
17      public ComponentDemo():this(100, 200, 400, 200){ }
18
19      private void InitializeComponents(){
20         _label1 = new Label();
21         _label1.Text = "This is a Label!";
22         _label1.Location = new Point(25, 25);
23
24         _button1 = new Button();
25         _button1.Text = "Click Me!";
26         _button1.Location = new Point(125, 25);
27         _button1.Click += new EventHandler(ButtonClickHandler);
28
29         _textbox1 = new TextBox();
30         _textbox1.Text = "some default text";
31         _textbox1.Location = new Point(225, 25);
32
33         this.Controls.Add(_label1);
34         this.Controls.Add(_button1);
35         this.Controls.Add(_textbox1);
36      }
37
38
39      public void ButtonClickHandler(Object sender, EventArgs e){
40         _label1.Text = _textbox1.Text;
41      }
42
43
44      public static void Main(){
45         Application.Run(new ComponentDemo());
46      } // end Main()
47   } // end class
```

Referring to Example 12.6 — I made only two minor modifications to the previous program. The first addition appears on line 27 where an event handler is registered with the _button1.Click event. Notice the use of the '+=' and new operators. Note that Click events require event handler methods with a signature of the EventHandler delegate type. The ButtonClickHandler() method, defined starting on line 39, sports the required signature, namely, it takes two parameters — one of type Object and the other of type EventArgs. The names of the method and its parameters can be just about anything you can think of, but I recommend keeping the parameter names the same, and choose a method name that makes it easy to identify it as an event handler method. In this example, I chose the name ButtonClickHandler.

Notice in the body of the ButtonClickHandler() method how the text box text is assigned to the label text. Let's see this program in action. Figure 12-12 gives the results of running the code. Referring to Figure 12-12 — the top image shows the state of affairs when the program first executes. When you click the button the label text changes to say "some default text", which is the text initially loaded into the tex tbox. In the bottom image the text "Events are cool!" is entered into the text box, the button clicked, and the label's text changed again.

 © 2008 Rick Miller — All Rights Reserved

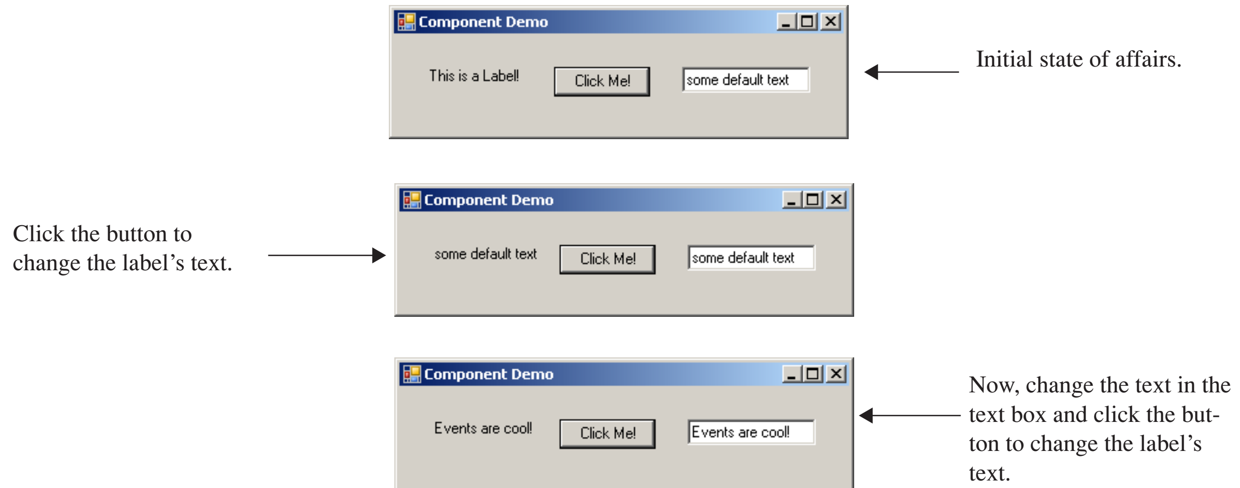


Figure 12-12: Results of Running Example 12.6 with Different Text in the TextBox

Quick Review

The whole point of creating a GUI is to have it respond to user interaction. All System.Windows.Forms GUI controls have event members. An *event* is something an object can respond to. For example, a Button can respond to a mouse click via its Click event.

A *delegate* declares a new type in the form of a method signature. Events are class members declared to have a certain delegate type, meaning that a method assigned to handle that event must have the specified delegate's method signature.

Use the '+=' operator to assign an event handler method to a control's event. Give your event handler method's names that clarify their role as event handlers.

Handling GUI Component Events In Separate Objects

In this section I am going to teach you a most critical skill, one that will liberate your thinking and take your programming skills to new heights. I'm going to show you how to handle GUI component events in separate objects. To do this you'll need to know how to do the following things:

- Create a stand-alone, non-application GUI class that extends Form.
- Create a separate application class that uses the services of the GUI class. This application class will also contain the required event handler code.
- Create GUI class constructors that take a reference to the object that contains the event handler code.
- Register a component's event with the event handler code via the supplied reference.
- Create appropriate methods or properties in the GUI class that allow horizontal manipulation of private GUI components.

Figure 12-13 gives the UML class diagram for the sample program I'm going to use to show you how to do these things. Referring to Figure 12-13 — the MyGUI class extends Form. I've shown the constructors for the MyGUI class. Note that one of the parameters in each constructor is of type MainApp. It's through this parameter that the MainApp class passes an instance of itself when it creates an instance of MyGUI. The MyGUI class then uses the MainApp reference to access the MainApp.ButtonClickHandler() method.

The MainApp class pulls double duty in this example. It contains the Main() method and some event handler code in the form of the ButtonClickHandler() method.

Let's now take a look at the code for these two classes. Example 12-7 gives the code for the MyGUI class.

12.7 MyGUI.cs

© 2008 Rick Miller — All Rights Reserved

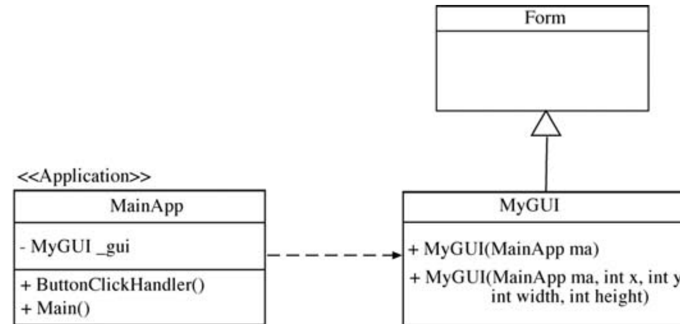


Figure 12-13: UML Class Diagram Showing Separate GUI and Application/Event Handler Classes

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4
5
6    public class MyGUI : Form {
7
8       /** Private GUI Components **/
9       private Button _button1;
10      private TextBox _textbox1;
11      private Label _label1;
12
13      /** Public Properties **/
14      public string TextBoxText {
15         get { return _textbox1.Text; }
16         set { _textbox1.Text = value; }
17      }
18
19      public string LabelText {
20         get { return _label1.Text; }
21         set { _label1.Text = value; }
22      }
23
24      /** Constructors **/
25      public MyGUI(MainApp ma, int x, int y, int width, int height){
26         this.Bounds = new Rectangle(x, y, width, height);
27         this.Text = "MyGUI Window";
28         InitializeComponents(ma);
29      }
30
31      public MyGUI(MainApp ma):this(ma, 100, 200, 400, 200){ }
32
33      /** Other Methods **/
34      private void InitializeComponents(MainApp ma){
35         _label1 = new Label();
36         _label1.Text = "This is a Label!";
37         _label1.Location = new Point(25, 25);
38
39         _button1 = new Button();
40         _button1.Text = "Click Me!";
41         _button1.Location = new Point(125, 25);
42         _button1.Click += new EventHandler(ma.ButtonClickHandler);
43
44         _textbox1 = new TextBox();
45         _textbox1.Text = "some default text";
46         _textbox1.Location = new Point(225, 25);
47
48         this.Controls.Add(_label1);
49         this.Controls.Add(_button1);
50         this.Controls.Add(_textbox1);
51      }
52   } // end MyGUI class definition
```

Referring to Example 12.7 — the MyGUI class is very similar in structure to Example 12.6. The primary difference is that it's no longer an application because I removed the Main() method. I have removed the ButtonClickHandler() method and moved it to the MainApp class. I have also added two public properties named *TextBoxText* and *LabelText* that allow access to the Text properties of the private _textbox1 and _label1 components. I also modified the two constructors by adding a MainApp type parameter, and have added a parameter of type MainApp to the Ini-

 © 2008 Rick Miller — All Rights Reserved

tializeComponents() method as well. Before I walk you through the operation of this code, take a look at the MainApp code given in Example 12.8.

12.8 MainApp.cs

```
1    using System;
2    using System.Windows.Forms;
3
4    public class MainApp {
5
6       private MyGUI _gui;
7
8       public MainApp(){
9          _gui = new MyGUI(this);
10         Application.Run(_gui);
11      }
12
13      public void ButtonClickHandler(Object sender, EventArgs e){
14           _gui.LabelText = _gui.TextBoxText;
15      }
16
17      public static void Main(){
18           new MainApp();
19      } // end Main()
20   } // end MyApp class definition
```

Referring to Example 12.8 — the MainApp class has a private field of type MyGUI named *_gui*. The MainApp constructor creates an instance of MyGUI and passes to its constructor a reference to itself via the `this` keyword. It then displays the window with a call to the Application.Run() method passing in the `_gui` reference. All the Main() method does is create an instance of MainApp.

The ButtonClickHandler() method shown on line 13 manipulates the text box and label text via the _gui reference by accessing the two public properties named TextBoxText and LabelText. Note that if you tried to access the MyGUI's _textbox1 and _label1 components directly, you'd get a compiler error because they are private fields, hence the need for public methods or properties to perform the required manipulations.

Let's now walk though the creation of the MyGUI object. When the MainApp constructor calls the MyGUI constructor, it passes in a reference to itself (*i.e.*, a reference to the object that's currently being created) via the `this` pointer. The single-parameter MyGUI constructor takes the reference and passes it on to the five parameter version of the MyGUI constructor. The `x`, `y`, `width`, and `height` parameters set the window's bounds. The `ma` parameter is passed as an argument to the InitializeComponents() method, where it is used to register its ButtonClickHandler() method with the _button1 Click event.

This is some of the most complex code you've encountered so far in this book. And though at this point it may seem difficult to trace through its execution, keep at it until you understand exactly what's happening in the code. Handling GUI events in separate objects is truly a critical programming skill and is a stepping stone to understanding and applying more complex object-oriented programming patterns in your code.

Now, let's see this bad boy run! Figure 12-14 shows the results of running this program.

Quick Review

Knowing how to write code so that GUI events generated in one object are handled by event handler methods located in another object is a critical programming skill. To do this, you must know how to do the following things: 1) create a stand-alone, non-application GUI class that extends Form, 2) create a separate application class that uses the services of the GUI class; this application class will also contain the required event handler code, 3) create GUI class constructors that take a reference to the object that contains the event handler code, 4) register a component's event with the event handler code via the supplied reference, and 5) create appropriate methods or properties in the GUI class that allow horizontal manipulation of private GUI components.

Layout Managers

As you saw earlier, placing GUI components in a window at absolute positions is tedious at best. While there may be times when you really want to put some component in a particular spot and have it stay there, it's generally preferable to make the window's components adjust their positions automatically to accommodate window resizing

© 2008 Rick Miller — All Rights Reserved

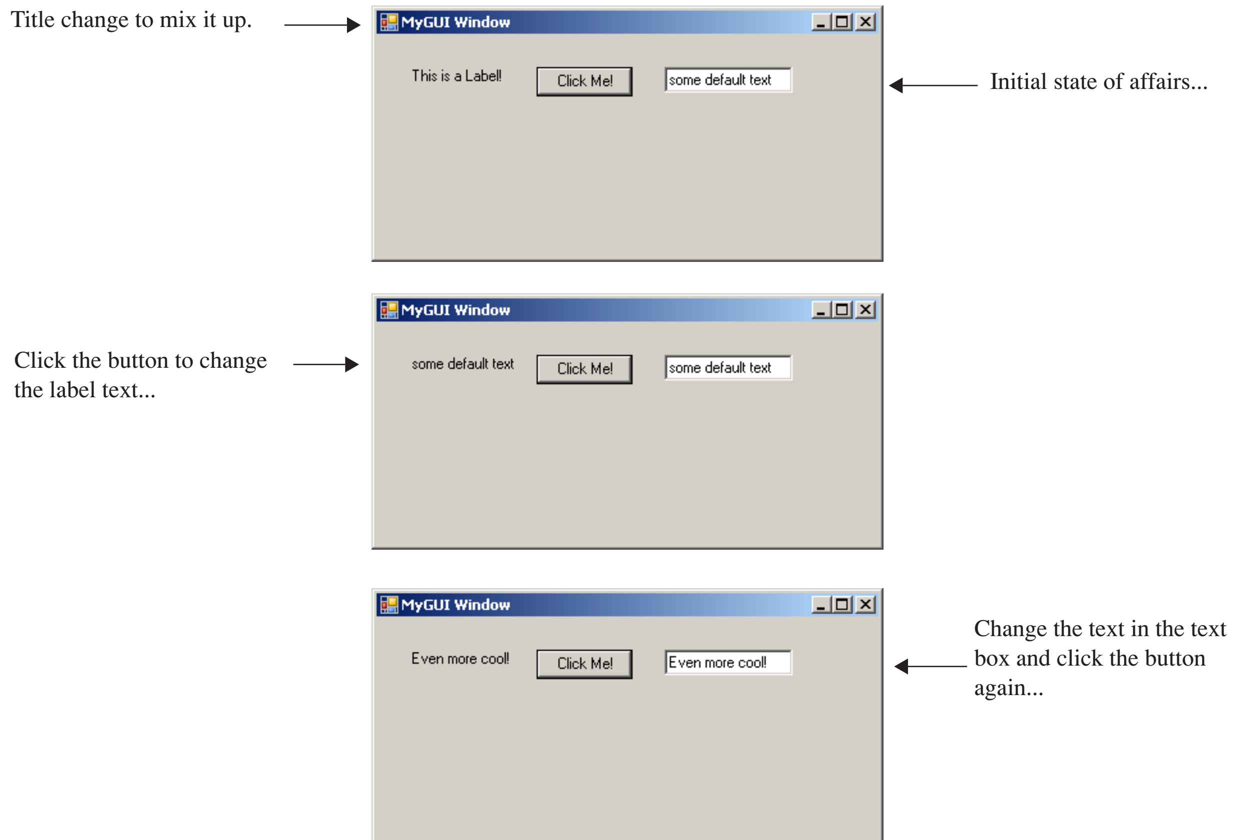


Figure 12-14: Results of Running Example 12.8 — GUI Events Handled in Separate Object

and prevent component clipping or hiding. In this section, I will show you how to automatically place components on a window via layout panels.

The .NET Framework provides two layout panels: *FlowLayoutPanel* and *TableLayoutPanel*. This may not seem like much, but you really can create complex window layouts using only these two layout panels.

FlowLayoutPanel

The purpose of the FlowLayoutPanel is to dynamically lay out its components either horizontally or vertically. When you resize a window that contains components in a FlowLayoutPanel, those components will float within the panel and readjust their position based on the size of the window. Let's see a FlowLayoutPanel in action.

Examples 12.9 and 12.10 give the code for a program that displays five buttons in a window. The buttons are placed in a FlowLayoutPanel. To make the program more interesting, I have added the capability for each button to display the time it was clicked relative to program launch.

12.9 FlowLayoutGUI.cs

```
1     using System;
2     using System.Windows.Forms;
3     using System.Drawing;
4
5     public class FlowLayoutGUI : Form {
6
7        private Button[] _buttonArray;
8        private FlowLayoutPanel _panel;
9
10       public FlowLayoutGUI(MainApp ma, int x, int y, int width, int height){
11          this.Bounds = new Rectangle(x, y, width, height);
12          this.Text = "Flow Layout GUI";
13          InitializeComponents(ma);
14       }
15
16       public FlowLayoutGUI(MainApp ma):this(ma, 100, 200, 425, 150){ }
```

 © 2008 Rick Miller — All Rights Reserved

```
17
18     public void InitializeComponents(MainApp ma){
19        _panel = new FlowLayoutPanel();
20        _panel.SuspendLayout();
21        _panel.AutoSize = true;
22        _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
23        _panel.WrapContents = true;
24        _panel.Dock = DockStyle.Top;
25
26        _buttonArray = new Button[5];
27
28        for(int i=0; i<_buttonArray.Length; i++){
29           _buttonArray[i] = new Button();
30           _buttonArray[i].Text = "Button " + (i+1);
31           _buttonArray[i].Click += new EventHandler(ma.ButtonClickHandler);
32           _panel.Controls.Add(_buttonArray[i]);
33        }
34
35        this.SuspendLayout();
36        this.Controls.Add(_panel);
37
38        _panel.ResumeLayout();
39        this.ResumeLayout();
40     }
41  } // end FlowLayoutGUI class definition
```

Referring to Example 12.9 — the FlowLayoutGUI class declares an array of Buttons and one FlowLayoutPanel. Most of the action takes place in the InitializeComponents() method. First, the FlowLayoutPanel is created. Several of its properties are then set, including *AutoSize*, *AutoSizeMode*, *WrapContents*, and *Dock*. Note the call to _panel.SuspendLayout() on line 20. Call the SuspendLayout() method anytime you are modifying several control properties at a time or are adding multiple controls to a control. The SuspendLayout() method suspends the target control's layout logic for improved performance. A call to SuspendLayout() must eventually be followed by a call to ResumeLayout().

The Dock property gets or sets how a control's edges are docked to its containing control and determines how it is resized. Note the use of the DockStyle enumeration to specify which edge to dock. The complete set of DockStyle values include DockStyle.*Bottom*, DockStyle.*Fill*, DockStyle.*Left*, DockStyle.*None*, DockStyle.*Right*, and DockStyle.*Top* as I have used here.

The button array is created on line 26 followed by a `for` loop that creates and initializes each button. Note that each button is added to the FlowLayoutPanel's Controls array. The buttons will be "flowed" into the FlowLayoutPanel from left to right by default in the order in which the buttons are added. You can change this behavior by setting the panel's *FlowDirection* property with the help of the FlowDirection enumeration whose values include FlowDirection.*BottomUp*, FlowDirection.*LeftToRight*, FlowDirection.*RightToLeft*, and FlowDirection.*TopDown*.

After all the buttons have been added to the FlowLayoutPanel, it's time to add the _panel reference to the window's Controls collection. This is done on line 36, and is preceded by a call to the window's SuspendLayout() method. Lastly, the ResumeLayout() method is called on both the _panel and the window.

Example 12.10 gives the code for the MainApp class.

12.10 MainApp.cs

```
1   using System;
2   using System.Windows.Forms;
3
4   public class MainApp {
5
6      private FlowLayoutGUI _gui;
7      private DateTime _appStart;
8
9      public MainApp(){
10        _gui = new FlowLayoutGUI(this);
11        _appStart = DateTime.Now;
12        Application.Run(_gui);
13     }
14
15     public void ButtonClickHandler(Object sender, EventArgs e){
16           ((Button)sender).Text = (DateTime.Now - _appStart).ToString();
17     }
18
19     public static void Main(){
20           new MainApp();
21     } // end Main()
22  } // end MyApp class definition
```

Referring to Example 12.10 — the MainApp class declares a FlowLayoutGUI field named _gui and a DateTime field named _appStart. In the MainApp constructor, the _gui reference is initialized to an instance of the FlowLayout-

© 2008 Rick Miller — All Rights Reserved

GUI class. The _appStart reference is initialized to DateTime.*Now,* which is simply a DateTime structure that represents the current date and time. The _appStart is then used in the body of the ButtonClickHandler() method to calculate the time span between the time the application started and the time the button was clicked. Note how the sender parameter is used to figure out which control generated the event. The result of subtracting one DateTime object from another results in a *TimeSpan* object. Every time a button in the GUI is clicked, its text is updated with the elapsed time the application has been running. Figure 12-15 shows the results of running this program.

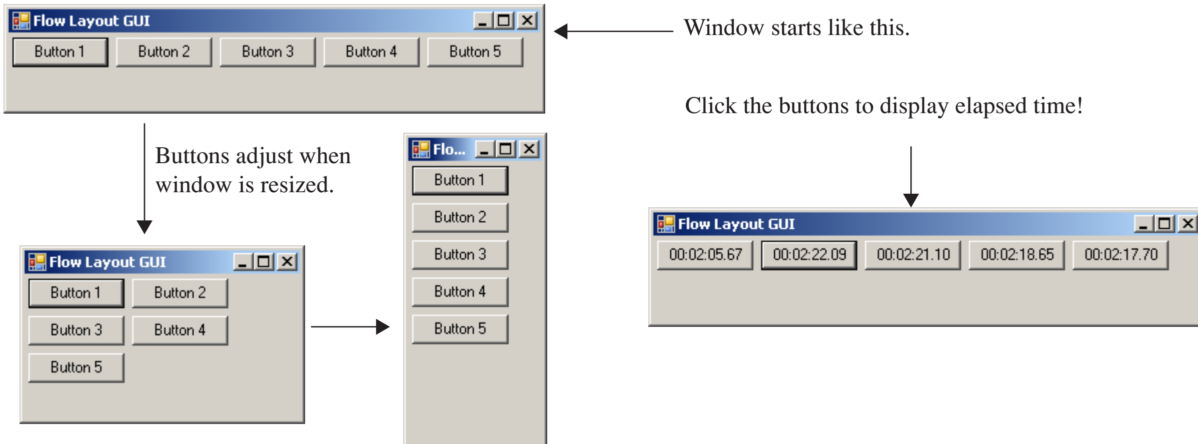


Figure 12-15: Results of Running Example 12.10 — Buttons Adjust when Window is Resized

TableLayoutPanel

The TableLayoutPanel lets you divvy up a panel into cells arranged by rows and columns. The order in which controls are added to a TableLayoutPanel is, by default, left-to-right and top-to-bottom. For example, if you create a TableLayoutPanel with two rows that each contain two columns and you add four buttons to it, the first button will go into cell 0,0, the second onto cell 0,1, the third into cell 1,0, and the last into cell 1,1.

The following program adds a slew of buttons to a TableLayoutPanel. When each button is clicked, its BackColor property is changed along with its Image. The program consists of both Examples 12.11 and 12.12.

12.11 TableLayoutGUI.cs

```
1     using System;
2     using System.Windows.Forms;
3     using System.Drawing;
4
5     public class TableLayoutGUI : Form {
6
7        Button[,] _floor = new Button[10,10];
8        TableLayoutPanel _panel;
9
10       public TableLayoutGUI(MainApp ma){
11         InitializeComponents(ma);
12        }
13
14    public void InitializeComponents(MainApp ma){
15         _panel = new TableLayoutPanel();
16         _panel.SuspendLayout();
17           _panel.ColumnCount = 10;
18         _panel.RowCount = 10;
19         _panel.Dock = DockStyle.Top;
20           _panel.AutoSize = true;
21         _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
22
23         for(int i = 0; i<_floor.GetLength(0); i++){
24           for(int j = 0; j<_floor.GetLength(1); j++){
25             _floor[i,j] = new Button();
26             _floor[i,j].Click += new EventHandler(ma.MarkSpace);
27             _panel.Controls.Add(_floor[i,j]);
28           }
29         }
30
31        this.SuspendLayout();
32        this.Text = "TableLayoutGUI Window";
```

 © 2008 Rick Miller — All Rights Reserved

```
33       this.Width = 850;
34       this.Height = 325;
35       this.Controls.Add(_panel);
36       _panel.ResumeLayout();
37       this.ResumeLayout();
38      } // end InitializeComponents() method
39   } // end TableLayoutGUI class definition
```

Referring to Example 12.11 — this program declares and creates a two-dimensional array of buttons having 10 rows and 10 columns. In the InitializeComponents() method, the TableLayoutPanel is created and its *RowCount* and *ColumnCount* properties are set to 10 x 10 to match the array's dimensions. The buttons are created in the nested `for` loop that starts on line 23, and added to the panel's Controls collection. Each button's Click event calls the MarkSpace() event handler method located in the MainApp class. Example 12.12 gives the code for the MainApp class.

12.12 MainApp.cs

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4
5    public class MainApp {
6
7       private TableLayoutGUI _gui;
8       private Bitmap _bitmap;
9       public MainApp(){
10         _gui = new TableLayoutGUI(this);
11         _bitmap = new Bitmap("rat.gif");
12         Application.Run(_gui);
13      }
14
15      public void MarkSpace(Object sender, EventArgs e){
16           ((Button)sender).BackColor = Color.Blue;
17           ((Button)sender).Image = _bitmap;
18       }
19
20      public static void Main(){
21         new MainApp();
22      } // end Main()
23   } // end MainApp class definition
```

Referring to Example 12.12 — the MainApp class declares a TableLayoutGUI field named _gui and a BitMap field named _bitmap. The MainApp constructor initializes the _gui and _bitmap references. The image used in this example is named "rat.gif" and is expected to be in the program's execution directory. (**Note:** The rat.gif image can be downloaded from the PulpFreePress.com or Warrenworks.com websites, or you can use your own image.)

When a button is clicked in the TableLayoutGUI window, the MarkSpace() method sets its BackColor and Image properties. Figure 12-16 shows the results of running this program.

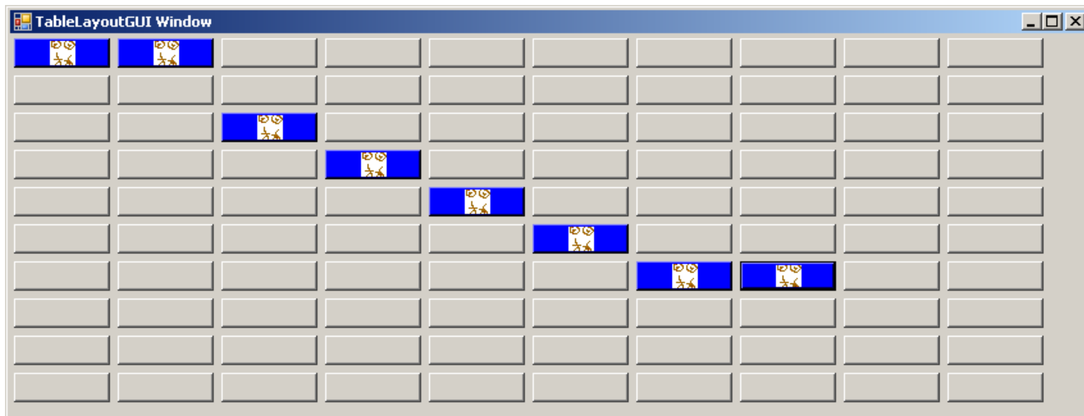


Figure 12-16: Results of Running Example 12.12 after several Buttons have been Clicked

Quick Review

Use the FlowLayoutPanel and TableLayoutPanel classes to automatically control the layout of controls in a window. Controls placed in a FlowLayoutPanel flow into the panel from left-to-right by default. Don't forget to set the panel's Dock property.

 © 2008 Rick Miller — All Rights Reserved

The TableLayoutPanel places controls into a grid arrangement, where each control occupies a cell that can be accessed via x and y coordinates, much like a two-dimensional array.

You can add panels to panels to create complex GUIs.

Menus

Professional looking GUIs are usually controlled via menus. In this section, I am going to show you how to add menus to your GUIs.

The .NET Framework provides several classes that make adding menus easy. These include the System.Windows.Forms.*MenuStrip* and the System.Windows.Forms.*ToolStripMenuItem*. The example program used to demonstrate the operation of these classes allows the user to dynamically add buttons and text boxes to a window via a menu. The Add menu contains three menu items named Button, TextBox, and Exit. It also contains a menu item separator. Figure 12-17 shows the application's window and menu structure.

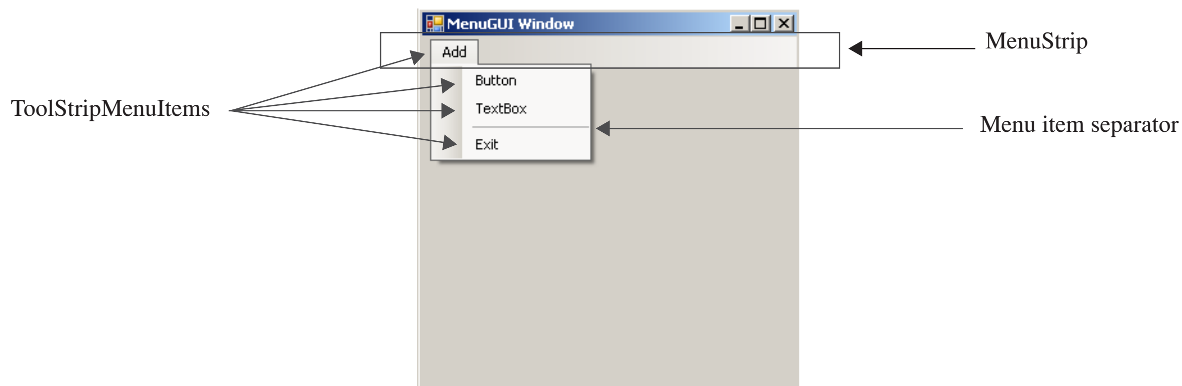


Figure 12-17: Window and Menu Structure of Menu Demo Program

Referring to Figure 12-17 — the MenuStrip control is docked at the top of the MenuGUI window. The Add menu item is added to the MenuStrip, and the Button, TextBox, item separator, and Exit menu items are added as submenu items to the Add menu item. Example 12.13 gives the code for this window.

12.13 MenuGUI.cs

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4
5    public class MenuGUI : Form {
6
7      private FlowLayoutPanel _panel;
8
9
10     public MenuGUI(MainApp ma, int x, int y, int width, int height){
11       this.Bounds = new Rectangle(x, y, width, height);
12       this.Text = "MenuGUI Window";
13       InitializeComponents(ma);
14     }
15
16     public MenuGUI(MainApp ma):this(ma, 125, 125, 300, 300){ }
17
18
19     private void InitializeComponents(MainApp ma){
20       MenuStrip ms = new MenuStrip();
21
22       ToolStripMenuItem addMenu = new ToolStripMenuItem("Add");
23       ToolStripMenuItem addButtonItem = new ToolStripMenuItem("Button", null,
24                                                         new EventHandler(ma.AddButtonItemHandler));
25       ToolStripMenuItem addTextBoxItem = new ToolStripMenuItem("TextBox", null,
26                                                         new EventHandler(ma.AddTextBoxItemHandler));
```

 © 2008 Rick Miller — All Rights Reserved

```
27        ToolStripMenuItem addExitItem = new ToolStripMenuItem("Exit", null,
28                                                            new EventHandler(ma.AddExitItemHandler));
29
30        addMenu.DropDownItems.Add(addButtonItem);
31        addMenu.DropDownItems.Add(addTextBoxItem);
32        addMenu.DropDownItems.Add("-"); // <---- use a dash to add menu item separators
33        addMenu.DropDownItems.Add(addExitItem);
34
35        ms.Items.Add(addMenu);
36        ms.Dock = DockStyle.Top;
37        this.MainMenuStrip = ms;
38
39        _panel = new FlowLayoutPanel();
40       _panel.SuspendLayout();
41        _panel.AutoSize = true;
42        _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
43        _panel.WrapContents = true;
44        _panel.Dock = DockStyle.Fill;
45
46
47        this.SuspendLayout();
48        this.Controls.Add(_panel);
49        this.Controls.Add(ms); // IMPORTANT: Add the MenuStrip last !!
50        _panel.ResumeLayout();
51        this.ResumeLayout();
52
53     }
54
55     public void AddButton(MainApp ma){
56        Button b = new Button();
57        b.Text = "C# Rocks";
58        b.Click += new EventHandler(ma.ButtonClickHandler);
59        _panel.SuspendLayout();
60        _panel.Controls.Add(b);
61        _panel.ResumeLayout();
62     }
63
64
65     public void AddTextBox(MainApp ma){
66        TextBox t = new TextBox();
67        t.Text = "Default Text";
68        t.Click += new EventHandler(ma.TextBoxClickHandler);
69        _panel.SuspendLayout();
70        _panel.Controls.Add(t);
71        _panel.ResumeLayout();
72     }
73  } // end MenuGUI class definition
```

Referring to Example 12.13 — the MenuGUI class declares one private FlowLayoutPanel field named _panel. All of the menu items are declared locally within the InitializeComponents() method. You may be asking yourself at this point why not all of the GUI components are declared as fields. The answer depends on which components you need to have access to after the GUI is rendered. In this example, I am adding buttons and text boxes to the flow layout panel. If I were designing an application that needed to add menu items to the menu strip, then I would most likely declare a MenuStrip field for easy access.

Let's step through the InitializeComponents() method. The first thing I do on line 20 is to declare and create the MenuStrip. Next, on lines 22 through 28, I declare and create the four ToolStripMenuItems. Notice the naming convention I have adopted here to help sort out which sub-items belong to which parent menu item.

The ToolStripMenuItem constructor is overloaded. I have used two versions in this code. The first version used on line 22 takes the name of the menu item. The second type of constructor takes three arguments: *name*, *image*, and *event handler*. I've used "null" to indicate no image. Notice how, by supplying an event handler, menu items are enabled to perform work.

Next, on lines 30 through 33, I add the submenu items to the addMenu item by adding them with its DropDownItems.Add() method. Notice here how a menu item separator is added to a list of menu items by adding a dash "-".

Once the submenu items are added to the parent menu item, the parent menu item is added to the MenuStrip by calling its Items.Add() method. The MenuStrip is docked to the top of the window and then designated as the MenuStrip for this window.

Lines 39 through 44 prepare the FlowLayoutPanel by setting several of its properties. Then, on line 47 the window's layout is suspended with a call to SuspendLayout(), then the panel is added to the window first, followed by the MenuStrip. (**Note:** Always add the MenuStrip last to ensure it displays at the top of the form and does not hide other controls.)

 © 2008 Rick Miller — All Rights Reserved

Lastly, the ResumeLayout() method is called on both the panel and the window.

The MenuGUI class contains two other public methods named AddButton() and AddTextBox(). To see these methods in action, let's take a look at the MainApp class given in Example 12.14.

12.14 MainApp.cs

```
1    using System;
2    using System.Windows.Forms;
3
4    public class MainApp {
5
6       private MenuGUI _gui;
7       private DateTime _appStart;
8
9       public MainApp(){
10        _gui = new MenuGUI(this);
11        _appStart = DateTime.Now;
12        Application.Run(_gui);
13      }
14
15      public void AddButtonItemHandler(object sender, EventArgs e){
16        _gui.AddButton(this);
17      }
18
19      public void AddTextBoxItemHandler(object sender, EventArgs e){
20        _gui.AddTextBox(this);
21      }
22
23      public void AddExitItemHandler(object sender, EventArgs e){
24        Application.Exit();
25      }
26
27      public void ButtonClickHandler(Object sender, EventArgs e){
28        ((Button)sender).Text = (DateTime.Now - _appStart).ToString();
29      }
30
31      public void TextBoxClickHandler(Object sender, EventArgs e){
32        ((TextBox)sender).Text = (DateTime.Now - _appStart).ToString();
33      }
34
35      public static void Main(){
36        new MainApp();
37      }
38   }
```

Referring to Example 12.14 — this version of the MainApp class declares two fields, one of type MenuGUI named _gui and the other of type DateTime named _appStart. Its constructor initializes the fields and kicks off the program with a call to Application.Run().

This class also contains five event handler methods, three of which are used by the menu items. The ButtonClick-Handler() method is used by all of the buttons that get added to the window, and the TextBoxClickHandler() method is used by all the text boxes.

When this program runs, users can add as many buttons or text boxes as they want by selecting the appropriate menu item. A click on either a button or a text box results in its text being set to the elapsed program run time. Figure 12-18 shows this program in action after several buttons and textboxes have been added to the window and then clicked.

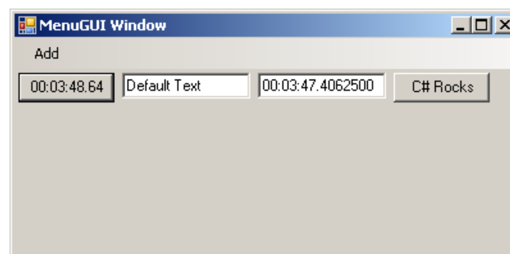


Figure 12-18: Results of Running Example 12.14 and Adding Several Buttons and Text Boxes

 © 2008 Rick Miller — All Rights Reserved

Quick Review

Use menus to give your GUI a professional appearance. You can create menus using the MenuStrip and ToolStripMenuItem classes.

A Little More About TextBoxes

Up until now, I've only used TextBox controls in single line mode ideally suited to display one short line of text. The TextBox, however, is a versatile control that can be used to edit multiline text or rich text content. In this section, I want to show you how to display a multi-line TextBox and then, by double-clicking on a line of text in the box, determine the text line number. You'll find this to be a handy little trick to have up your sleeve.

The code for the example program is given in two files. Example 12.15 gives the code for the LineSelectGUI class and Example 12.16 gives the code for the MainApp class.

12.15 LineSelectGUI.cs

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4    using System.Windows;
5
6    public class LineSelectGUI : Form {
7
8       private TextBox _tb1;
9       private TextBox _tb2;
10      private FlowLayoutPanel _flowLayoutPanel;
11
12      public LineSelectGUI(MainApp ma, int x, int y, int width, int height){
13         this.Bounds = new Rectangle(x, y, width, height);
14         this.Text = "LineSelectGUI Window";
15         InitializeComponents(ma);
16       }
17
18       public LineSelectGUI(MainApp ma):this(ma, 125, 125, 375, 200){ }
19
20       public void InitializeComponents(MainApp ma){
21
22         _flowLayoutPanel = new FlowLayoutPanel();
23         _flowLayoutPanel.SuspendLayout();
24         _flowLayoutPanel.Height = 56;
25         _flowLayoutPanel.Width = 20;
26         _flowLayoutPanel.AutoSize = true;
27         _flowLayoutPanel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
28         _flowLayoutPanel.WrapContents = true;
29         _flowLayoutPanel.Dock = DockStyle.Top;
30
31         _tb1 = new TextBox();
32         _tb1.Multiline = true;
33         _tb1.Height = 150;
34         _tb1.Width = 200;
35         _tb1.DoubleClick += new EventHandler(ma.DoubleClickHandler);
36
37         _tb2 = new TextBox();
38          _flowLayoutPanel.Controls.Add(_tb1);
39          _flowLayoutPanel.Controls.Add(_tb2);
40
41         this.SuspendLayout();
42         this.Controls.Add(_flowLayoutPanel);
43
44         _flowLayoutPanel.ResumeLayout();
45         this.ResumeLayout();
46      }
47
48      public void ShowLineNumber(){
49          int index = _tb1.SelectionStart;
50          int line_number = _tb1.GetLineFromCharIndex(index);
51         _tb2.Text = ("Line Number is: " + line_number);
52       }
53
54   } // end LineSelectGUI class definition
```


© 2008 Rick Miller — All Rights Reserved

Referring to Example 12.15 — the LineSelectGUI class declares two TextBox fields named _tb1 and _tb2 and one FlowLayoutPanel field named _flowLayoutPanel. In the InitializeComponents() method the FlowLayoutPanel is created and initialized. Next, the first TextBox field, _tb1, is created and initialized to become a multiline TextBox by setting its *MultiLine* property to true. Setting its *WrapContents* property to true makes text wrap automatically to the next line. Finally, on line 29, the MainApp.DoubleClickHandler() method is registered with _tb1's Click event.

The second TextBox is created on line 37, and on lines 38 and 39 both TextBoxes are added to the FlowLayout-Panel.

The LineSelectGUI class defines a method named ShowLineNumber() starting on line 48. The ShowLineNumber() method determines the text line number with the help of two TextBox methods. First, the index of the selected text must be determined by calling the TextBox.SelectionStart() method. The index of the selected text is then used in a call to the TextBox.GetLineFromCharIndex() method.

The ShowLineNumber() method is called within the body of the MainApp.DoubleClickHandler() method. Example 12.16 gives the code for the MainApp class.

12.16 MainApp.cs

```
1   using System;
2   using System.Windows.Forms;
3
4   public class MainApp {
5
6     private LineSelectGUI _gui;
7
8     public MainApp(){
9       _gui = new LineSelectGUI(this);
10      Application.Run(_gui);
11    }
12
13    public void DoubleClickHandler(Object sender, EventArgs args){
14       _gui.ShowLineNumber();
15     }
16
17    public static void Main(){
18      new MainApp();
19    }
20  }
```

Referring to Example 12.16 — the MainApp class declares one field of type LineSelectGUI named _gui. The MainApp constructor initializes the _gui field and passes it as an argument to the Application.Run() method. A double-click within the multiline text box results in a call to the DoubleClickHandler() method, which in turn calls the _gui.ShowLineNumber() method. Figure 12-19 shows the results of running this program.

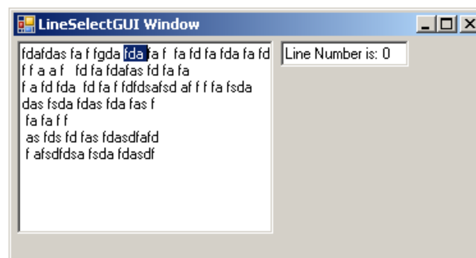


Figure 12-19: Results of Running Example 12.16 — Double-clicking the First Line

Quick Review

TextBoxes can be used in single-line or multiline mode. To create a multiline text box set its *Multiline* property to true. To determine the line number for a line of text in a text box, use the TextBox.SelectionStart() method to first get the index of the selected character, then use the TextBox.GetLineFromCharIndex() method to get the text line.

 © 2008 Rick Miller — All Rights Reserved

The Rhythm Of Coding GUIs

You may have noticed by now that there is a certain rhythm associated with writing GUI code by hand. Regardless of how complex you make your GUI, getting into the rhythm can make writing the code much less tedious and lots of fun. The rhythm goes something like this:

- Start by sketching a layout of your GUI. This will be a tremendous help when you start coding.
- Declare components like dialog windows, panels, menus, buttons, text boxes, labels, etc.
- Initialize the components in a constructor or in another method that's called by the constructor.
- If using absolute positioning, set the component's placement upon the window.
- Register event handlers.
- Set other component properties as required.
- Add components to panels.
- Add panels to a form.

Summary

The Form class, found in the System.Windows.Forms namespace, serves as the basis for all types of windows you might need to create in your application. These include standard, tool, borderless, or floating windows. The Form class is also used to create dialog boxes and multiple-document interface (MDI) windows. A Form is a ContainerControl, a ScrollableControl, a Control, a Component, a MarshalByRefObject, and ultimately an Object.

The Form class provides a lot of functionality right out of the box. You can drag the window around the screen, resize the window, minimize the window, maximize the window, and close the application by clicking the box with the "X" in the upper right corner.

Microsoft Windows applications are *event-driven*. When launched they wait patiently for an event such as a mouse click or keystroke to occur. Events are delivered to the application in the form of *messages*. Messages can be generated by the operating system in response to various types of stimuli, including direct user interaction (*i.e.,* mouse movement, clicking, scrolling, keystrokes, etc.), or by other applications.

The majority of system generated messages are placed into a data structure referred to as the *system message queue*. A *queue* is a data structure that has a first-in-first-out (FIFO) characteristic. Messages are placed into the queue as they occur. Each message contains data that includes the type of event, target window, cursor coordinates, etc. The system then examines each message and routes it to its target window.

There are two types of coordinates: *screen coordinates* and *window coordinates*. Window coordinates are also referred to as client coordinates. The basic unit of measure for a screen is the *pixel*. The origin of the screen, or the point where both the value of its x and y coordinates equal 0, is located in the upper left corner of the screen. The value of the x coordinate increases to the right, while the value of the y coordinate increases down. Points upon the screen are given in (x, y) pairs. Windows have a coordinate system similar to the screen, with their origin located in the upper left hand corner of the window. Windows, and the components drawn within them, have height and width. The bounds of a component are the location of its upper left corner together with its width and height.

The Form class provides many properties, methods, and events which make it easy to manipulate them in your programs. The Form class gets most of its functionality via its inheritance hierarchy. What you can do to a form you can also do, for the most part, to other controls, containers, and components.

Setting a control's properties often requires the use of other classes, structures, or enumerations found in the .NET Framework. A few of these include System.Drawing.Point, System.Drawing.Rectangle, System.Drawing.Color, System.Drawing.Bitmap, and System.Drawing.Image. The type of property will determine what type of object you must use to set the property.

To add a control like a Button or TextBox to a window, you must first declare and create the control, set its properties, and then add the control to the window's Controls collection. The absolute placement of controls can be tedious. Use the System.Drawing.Rectangle class to set a control's Bounds property. You may alternatively set a control's Top, Left, Width, and Height properties separately.

© 2008 Rick Miller — All Rights Reserved

The whole point of creating a GUI is to have it respond to user interaction. All System.Windows.Forms GUI controls have Event members. An *event* is something an object can respond to. For example, a Button can respond to a mouse click via its Click event.

A *delegate* declares a new type in the form of a method signature. Events are class members declared to have a certain delegate type, meaning that a method assigned to handle that event must have the specified delegate's method signature.

Use the '+=' operator to assign an event handler method to a control's event. Give your event handler methods names that clarify their role as event handlers.

Knowing how to write code so that GUI events generated in one object are handled by event handler methods located in another object is a critical programming skill. To do this you, must know how to do the following things: 1) create a stand-alone, non-application GUI class that extends Form, 2) create a separate application class that uses the services of the GUI class; this application class will also contain the required event handler code, 3) create GUI class constructors that take a reference to the object that contains the event handler code, 4) register a component's event with the event handler code via the supplied reference, and 5) create appropriate methods or properties in the GUI class that allows horizontal manipulation of private GUI components.

Use the FlowLayoutPanel and TableLayoutPanel classes to automatically control the layout of controls in a window. Controls placed in a FlowLayoutPanel flow into the panel from left-to-right by default. Don't forget to set the panel's Dock property.

The TableLayoutPanel is used to place controls into a grid arrangement, where each control occupies a cell that can be accessed via x and y coordinates, much like a two-dimensional array.

You can add panels to panels to create complex GUIs.

Use menus to give your GUI a professional appearance. You can create menus using the MenuStrip and ToolStripMenuItem classes.

TextBoxes can be used in single-line or multiline mode. To create a multiline text box set its Multiline property to true. To determine the line number for a line of text in a text box use the TextBox.SelectionStart() method to first get the index of the selected character, then use the TextBox.GetLineFromCharIndex() method to get the text line.

Getting into the rhythm of writing GUI code can make writing the code much less tedious and lots of fun.

Skill-Building Exercises

1. **API Drill:** Visit the .NET API documentation located on the MSDN website and explore the System.Windows.Forms namespace. List and briefly describe the purpose of each class, structure, enumeration, and delegate.

2. **API Drill:** Visit the .NET API documentation located on the MSDN website and explore the System.Drawing namespace. List and briefly describe the purpose of each class, structure, enumeration, and delegate.

3. **API Drill:** Visit the .NET API documentation located on the MSDN website and research the System.Windows.Forms.Control class. List and briefly describe the purpose of each of its members.

4. **Code Drill:** Experiment with control docking. First, visit the MSDN website and research the purpose of the Control.Dock property. Write a short program that puts several buttons in a window. Set each button's Dock property using the DockStyle enumeration. Note the effects each different DockStyle location has upon the buttons.

5. **Code Drill:** Experiment with control anchoring. First, visit the MSDN website and research the purpose of the Control.Anchor property. Write a short program that puts several buttons in a window. Set each button's Anchor property using the AnchorStyles enumeration. Note the effects that different AnchorStyles have upon the buttons.

6. **Code Drill:** Write a program that creates 10 buttons, puts them into a FlowLayoutPanel, and displays them in a window. Experiment with setting the different FlowLayoutPanel properties and note the effects.

7. **Code Drill:** Practice creating complex GUI layouts by writing a program that uses a combination of FlowLayout-

 © 2008 Rick Miller — All Rights Reserved

Panels and TableLayoutPanels. Add buttons to each panel and then add one type of panel to another.

8. **Code Drill:** Draw a complex user interface on a napkin or piece of paper. Include buttons, single-line and multiline textboxes, and labels. Divide the user interface into different areas, where one area might have only the multiline text box and another area the buttons, and still another area the labels and single-line textboxes. When you finish your drawing write a program that displays the controls in a window according to your plan. You don't need to worry about responding to user events or functionality. This is just a control placement exercise.

9. **API/Code Drill:** Explore the System.Windows.Forms namespace and select several controls, like CheckBox or RadioButton for example, that weren't covered in this chapter. Research their functionality and write a short program that uses them in a window.

Suggested Projects

1. **Programming:** Revisit the Robot Rat project presented in Chapter 3 and give it a graphical user interface. You may take several approaches to this project. For example, you can represent the floor as an array of labels or buttons placed within a TableLayoutGrid. This grid of controls would appear in one section of the user interface. Another section of the user interface would contain a set of buttons that allowed users to control the robot rat's movements. Separate the user interface code from the event handler code. Another approach would be to move an image of a robot rat around a window. This would require you to research how to display images directly in a window and update the window via its Paint event when the image is moved.

2. **Programming:** Write a program that mimics the operation of a handheld calculator. At a minimum implement the add, subtract, multiply, and divide operations. You may lay out the calculator's user interface any way you want, but one approach would be to put the display in the top section and a grid of buttons in the bottom section. You might even have separate sections for the numbers and the function keys. Separate the user interface code from the event handler code.

Self-Test Questions

1. How many different types of windows can be created with the Form class?

2. How are operating system messages generated and sent to a GUI application?

3. How are controls added to forms?

4. What are the differences between *screen coordinates* and *client coordinates*?

5. What does the term *origin* mean?

6. What four pieces of data define the bounds of a control?

7. What's the purpose of a delegate type?

8. How are delegates and event-handler method signatures related? How are delegates and events related?

9. What operator do you use to assign an event handler method to a control's event?

10. Briefly describe the general steps required to respond to a control's event with an event handler located in a differ-

© 2008 Rick Miller — All Rights Reserved

ent object.

11. What's the purpose of the Control.SuspendLayout() method? What method should be called to resume layout?

12. What's the difference between the FlowLayoutPanel and TableLayoutPanel?

13. How can you change the direction in which controls flow into a FlowLayoutPanel?

14. How do controls flow into a TableLayoutPanel?

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

Notes

 © 2008 Rick Miller — All Rights Reserved

Chapter 13

Voigtlander Bessa-L / 15mm Super Wide-Heliar

Fairview Park

Custom Events

Learning Objectives

- List and describe the components required to implement custom events
- Describe the purpose and use of delegates
- Describe the purpose and use of events
- Use the "delegate" keyword to declare new delegate types
- Use the "event" keyword to declare new event members
- Create a custom event argument passing class
- Create event-argument objects and pass to event handler methods
- Describe the event processing cycle
- Create and use event publisher classes
- Create and use event subscriber classes

© 2008 Rick Miller — All Rights Reserved

Introduction

Graphical User Interface (GUI) components are not the only type of objects that can have event members. Indeed, you can add events to the classes you design, and that's the subject of this chapter.

To add custom events to your programs, you'll need to know a little something about the following topics: 1) how to use the *delegate* keyword to declare new delegate types, 2) how to use the *event* keyword to declare new event members using delegate types, 3) how to create a class that conveys event data between an event publisher and an event subscriber, 4) how to create an event publisher, 5) how to create an event subscriber, 6) how to create event handler methods, and 7) how to register event handlers with a particular event. If you have read Chapter 12, you already know how to do items 6 and 7 on the list.

The information and programming techniques you learn from reading this chapter will open up a whole new world of programming possibilities.

C# Event Processing Model: An Overview

C# is a modern programming language supported by an extensive Application Programming Interface (API) referred to as the .NET Framework. And, as you learned in Chapter 12, C# also supports event-driven programming normally associated with Microsoft Windows applications. One normally thinks of events as being generated exclusively by GUI components, but any object can generate an event if it's programmed to do so.

You need two logical components to implement the event processing model: 1) an event producer (or *publisher*), and 2) an event consumer (or *subscriber*). Each component has certain responsibilities. Consider the following diagram:

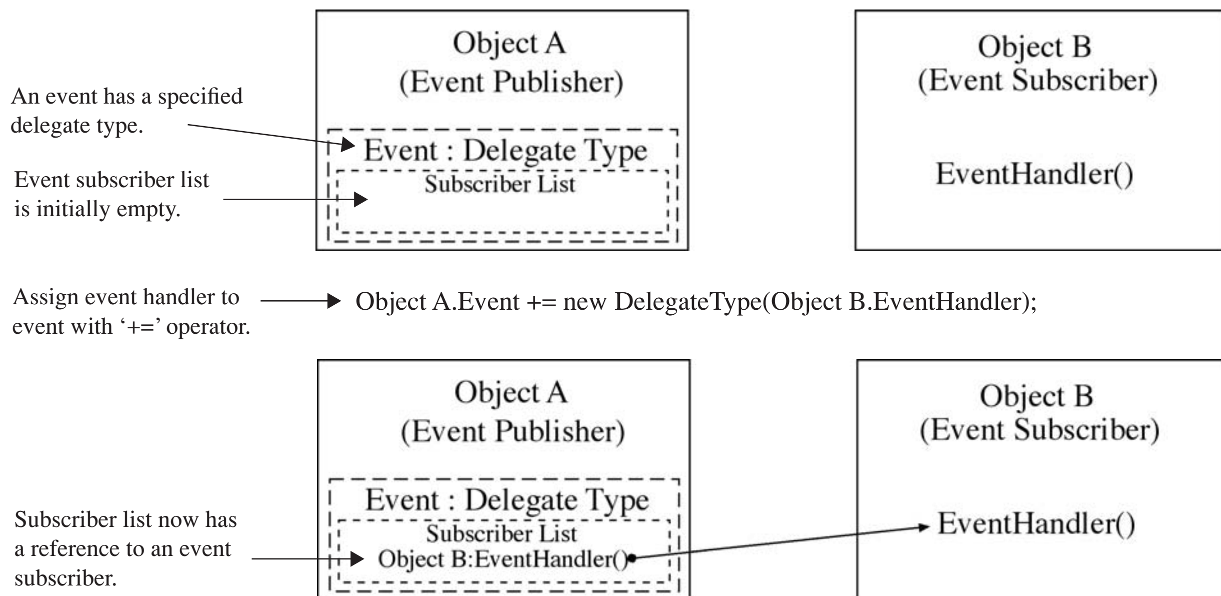


Figure 13-1: Event Publisher and Subscriber

Referring to Figure 13-1 — each event in Object A has a specified delegate type. The delegate type specifies the authorized method signature for event handler methods. However, the delegate does more than just specify a method signature; a delegate object maintains a list of event subscribers in the form of references to event handler methods. These references can point to an object and one of its instance methods or to a static method. The event's subscriber list is initially empty until the first subscriber has been added to it with the '+=' operator. The EventHandler() method defined in Object B must conform to the method signature specified by the event's delegate type. If you attempt to use an event handler method that does not conform to the delegate type's method signature, you will receive a compiler error. Let's now substitute some familiar names for Object A and Object B. Figure 13-2 offers a revised diagram.

© 2008 Rick Miller — All Rights Reserved

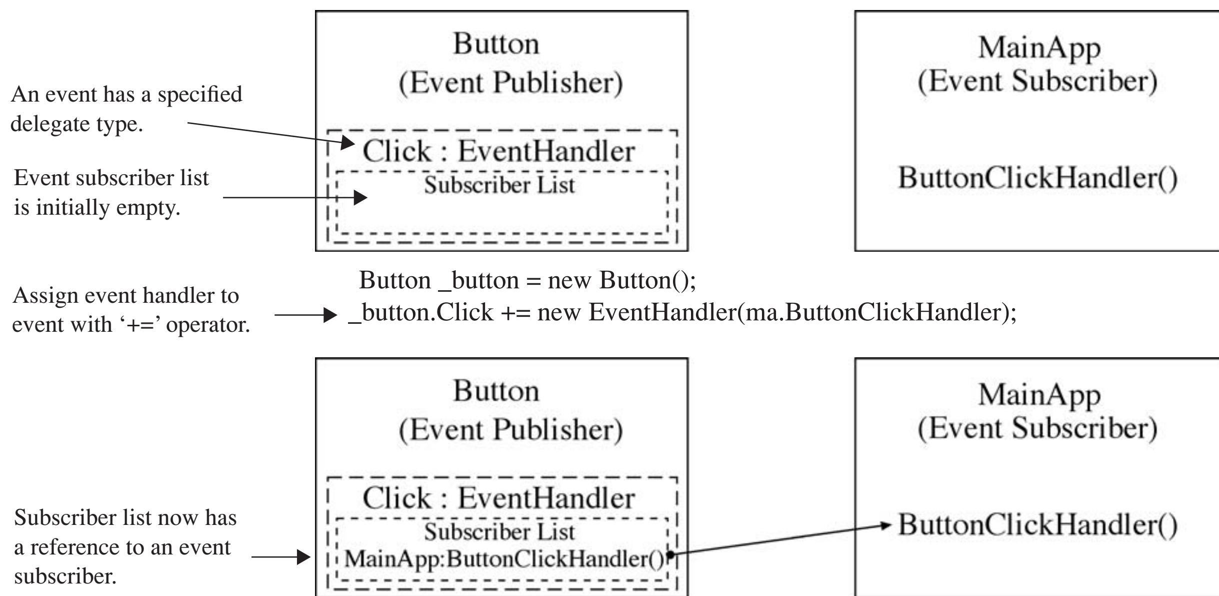


Figure 13-2: Event Publisher and Subscriber

Referring to Figure 13-2 — a button's Click event has an EventHandler delegate type. As you learned in Chapter 12, event handler methods assigned to a button's Click event must have the following signature:

```
void MethodName(object sender, EventArgs e)
```

The method's name can be pretty much anything you want it to be, but it must declare two parameters, the first of type *object*, and the second of type *EventArgs*, and it must return void. The names of the parameters can be anything you want as well, but the names *sender* and *e* work just fine.

When the button's Click event is fired, the Click event's delegate instance calls each registered subscriber's event handler method in the order it appears in the event subscriber list. This is all handled behind the scenes as you will soon see. In the case of a button and other controls, there exists an internal method that is called when a Click event occurs that kicks off the subscriber notification process. Remember that when talking about GUI components, a mouse click results in the generation of a message that is ultimately routed to the window in which the mouse click occurred. This message is translated into a Click event. When writing custom events, you can intercept messages and translate them into events or write a method that generates events out of thin air or in response to some other stimulus.

Quick Review

You need two logical components to implement the event processing model: 1) an event producer (*publisher*), and 2) an event consumer (*subscriber*). A *delegate* type specifies the authorized method signature for event handler methods. A delegate object maintains a list of event subscribers in the form of references to event handler methods. An event's subscriber list is initially empty until the first subscriber has been added to it with the '+=' operator. Event handler methods must conform to the method signature specified by an event's delegate type.

Custom Events Example: Minute Tick

The best way to get your head around custom events is to study an example application. This section presents a short program that implements custom events. The Minute Tick application consists of five source files, four of which appear in the Unified Modeling Language (UML) class diagram shown in Figure 13-3. Referring to Figure 13-3 — both the Publisher and Subscriber classes depend on the MinuteEventArgs class and the ElapsedMinuteEventHandler delegate. The Publisher class contains a MinuteTick event, which is of type ElapsedMinuteEventHandler. The

 © 2008 Rick Miller — All Rights Reserved

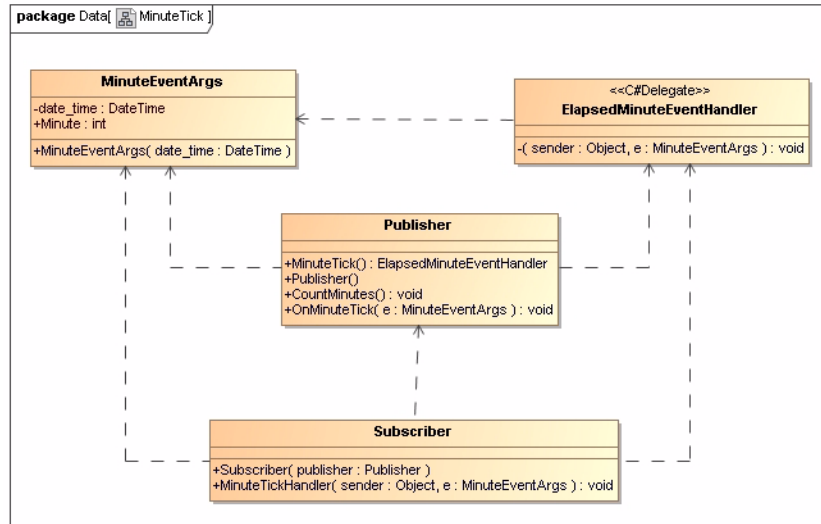


Figure 13-3: Minute Tick UML Class Diagram

ElapsedMinuteEventHandler delegate depends on the MinuteEventArgs class because it is the type of one of its parameters, as is shown in the diagram.

The complete Minute Tick application source code is given in Examples 13.1 through 13.5.

13.1 MinuteEventArgs.cs

```
1    using System;
2
3    public class MinuteEventArgs : EventArgs {
4      private DateTime date_time;
5
6      public MinuteEventArgs(DateTime date_time){
7        this.date_time = date_time;
8      }
9
10     public int Minute {
11       get { return date_time.Minute; }
12     }
13   }
```

Referring to Example 13.1 — the MinuteEventArgs class extends the EventArgs class and adds a private field named *date_time* and one public read-only property named *Minute,* which simply returns the value of the Minute property of the DateTime object.

13.2 ElapsedMinuteEventHandler.cs

```
1    using System;
2
3    public delegate void ElapsedMinuteEventHandler(Object sender, MinuteEventArgs e);
```

Referring to Example 13.2 — the ElapsedMinuteEventHandler delegate specifies a method signature that returns `void` and takes two parameters, the first one an object, and the second one of type MinuteEventArgs.

13.3 Publisher.cs

```
1    using System;
2
3    public class Publisher {
4
5       public event ElapsedMinuteEventHandler MinuteTick;
6
7
8       public Publisher(){
9         Console.WriteLine("Publisher Created");
10      }
11
12      public void CountMinutes(){
13        int current_minute = DateTime.Now.Minute;
14        while(true){
15          if(current_minute != DateTime.Now.Minute){
16             Console.WriteLine("Publisher: {0}", DateTime.Now.Minute);
17             OnMinuteTick(new MinuteEventArgs(DateTime.Now));
18             current_minute = DateTime.Now.Minute;
```

 © 2008 Rick Miller — All Rights Reserved

```
19          }//end if
20        } // end while
21      } // end CountMinutes method
22
23      public void OnMinuteTick(MinuteEventArgs e){
24         if(MinuteTick != null){
25            MinuteTick(this, e);
26          }
27      }// end OnMinuteTick method
28   } // end Publisher class definition
```

Referring to Example 13.3 — the Publisher class defines an event named *MinuteTick*. Notice that the MinuteTick event is of type ElapsedMinuteEventHandler. The CountMinutes() method that starts on line 12 contains a `while` loop that repeats forever and continuously compares the values of the current_minute with DateTime.Now.Minute. As soon as a change is detected in the two values, a brief message is written to the console followed by a call to the publisher's OnMinuteTick() method on line 17. Notice that when this method is called, a new MinuteEventArgs object is created and used as an argument to the method call. The OnMinuteTick() method definition begins on line 23. It takes the MinuteEventArgs parameter and passes it on to a call to the MinuteTick event. Note on line 24 how the `if` statement checks to see if the MinuteTick reference is null. It will be null if no event handler methods have been registered with the event.

13.4 Subscriber.cs

```
1    using System;
2
3    public class Subscriber {
4
5      public Subscriber(Publisher publisher){
6       publisher.MinuteTick += new ElapsedMinuteEventHandler(this.MinuteTickHandler);
7        Console.WriteLine("Subscriber Created");
8      }
9
10     public void MinuteTickHandler(Object sender, MinuteEventArgs e){
11       Console.WriteLine("Subscriber Handler Method: {0}", e.Minute);
12     }
13    } // end Subscriber class definition
```

Referring to Example 13.4 — the Subscriber class declares an event handler method on line 10 named *MinuteTickHandler()*. The MinuteTickHandler() method defines two arguments of the types required by the ElapsedMinuteEventHandler delegate type. The ElapsedMinuteEventHandler delegate is used on line 6 to register the subscriber's MinuteTickHandler() method with the publisher's MinuteTick event.

13.5 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4     public static void Main(){
5       Console.WriteLine("Custom Events are Cool!");
6
7       Publisher p = new Publisher();
8       Subscriber s = new Subscriber(p);
9       p.CountMinutes();
10
11    } // end main
12   } //end MainApp class definition
```

Referring to Example 13.5 — the MainApp class provides the Main() method. It simply creates a Publisher object and a Subscriber object, and then makes a call to the publisher's CountMinutes() method. Figure 13-4 shows the results of running this application. Note that the actual minutes displayed when the program runs depend on when you start the program.

Application started at 9 minutes past the hour. Your time outputs will reflect the time you run the program.

```
Projects - mainapp
C:\Documents and Settings\Rick\Desktop\Projects\Chapter13\MinuteTick>mainapp
Custom Events are Cool!
Publisher Created
Subscriber Created
Publisher: 9
Subscriber Handler Method: 9
Publisher: 10
Subscriber Handler Method: 10
Publisher: 11
Subscriber Handler Method: 11
Publisher: 12
Subscriber Handler Method: 12
```

Figure 13-4: Results of Running Example 13.5

© 2008 Rick Miller — All Rights Reserved

Custom Events Example: Automated Water Tank System

In this section, I want to show you how you might model a complex system using custom events. The system modeled here is a simple water tank that can be filled with water. Once the water reaches a certain level within the tank, a pump is activated and drains the tank until it again reaches a certain level. The tank contains two water level sensors. One acts as a high-level sensor and the other acts as a low-level sensor. The tank also has a pump that pumps water at a certain rate or pumping capacity. The system comprises the following classes: *Pump*, *WaterLevelEventArgs*, *WaterLevelEventHandler*, *WaterTank*, and *WaterSystemApp,* which serves as the main application class. Figure 13-5 gives the UML class diagram for the water tank system.

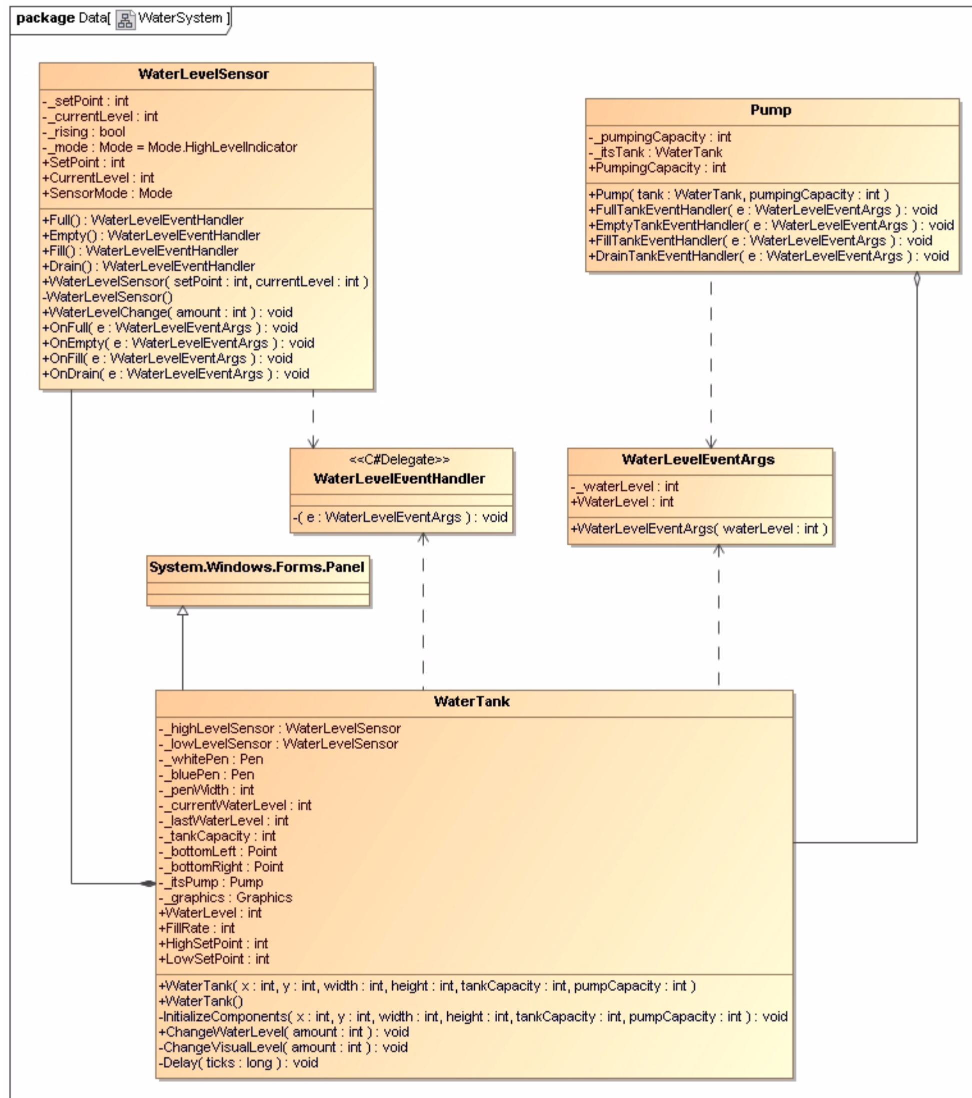


Figure 13-5: Water Tank System UML Class Diagram

Referring to Figure 13-5 — The WaterTank class extends the System.Windows.Forms.Panel class. This gives the water tank a visual representation. When water is added to the tank, the panel is filled in with blue lines as the water level rises. The lines are then overdrawn with a different color as the water level recedes. The WaterLevelEventArgs class is used to pass water level information between event publisher and subscriber. In this example, the WaterLevel-Sensor is the publisher and the Pump is the subscriber. The WaterLevelEventHandler delegate is used to declare the WaterLevelSensor’s Fill, Full, Drain, and Empty events. Let’s take a look at the code.

 © 2008 Rick Miller — All Rights Reserved

13.6 WaterLevelEventArgs.cs

```
1    using System;
2
3    public class WaterLevelEventArgs : EventArgs {
4
5      private int _waterLevel;
6
7      public WaterLevelEventArgs(int waterLevel){
8         WaterLevel = waterLevel;
9      }
10
11     public int WaterLevel {
12        get { return _waterLevel; }
13        set { _waterLevel = value; }
14     }
15   }
```

Referring to Example 13.6 — the WaterLevelEventArgs class contains one private integer field named *_waterLevel* and one public property named *WaterLevel*.

13.7 WaterLevelEventHandler.cs

```
1    using System;
2
3    public delegate void WaterLevelEventHandler(WaterLevelEventArgs e);
```

Referring to Example 13.7 — the WaterLevelEventHandler delegate specifies an event-handler method signature that returns `void` and contains one parameter of type WaterLevelEventArgs.

13.8 WaterLevelSensor.cs

```
1    using System;
2
3    public class WaterLevelSensor {
4       private int _setPoint;
5       private int _currentLevel;
6       private bool _rising;
7       private Mode _mode = Mode.HighLevelIndicator;
8
9
10      public enum Mode { HighLevelIndicator, LowLevelIndicator };
11      public event WaterLevelEventHandler Full;
12      public event WaterLevelEventHandler Empty;
13      public event WaterLevelEventHandler Fill;
14      public event WaterLevelEventHandler Drain;
15
16      public int SetPoint {
17         get { return _setPoint; }
18         set { _setPoint = value; }
19      }
20
21      public int CurrentLevel {
22         get { return _currentLevel; }
23         set { _currentLevel = value; }
24      }
25
26      public Mode SensorMode {
27         get { return _mode; }
28         set { _mode = value; }
29      }
30
31      public WaterLevelSensor(int setPoint, int currentLevel){
32         SetPoint = setPoint;
33         CurrentLevel = currentLevel;
34      }
35
36      private WaterLevelSensor(){ }
37
38      public void WaterLevelChange(int amount){
39         int lastLevel = CurrentLevel;
40         CurrentLevel += amount;
41         _rising = (CurrentLevel >= lastLevel);
42
43         switch(_mode){
44            case Mode.HighLevelIndicator :
45                 if(_rising){
46                   if(CurrentLevel >= SetPoint){
47                      WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
48                      OnFull(args);
49                    }else{
50                      WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
51                      OnFill(args);
```

 © 2008 Rick Miller — All Rights Reserved

```
52                  }
53               }
54               break;
55
56          case Mode.LowLevelIndicator :
57            if(!_rising){
58            if(CurrentLevel <= SetPoint){
59                WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
60                OnEmpty(args);
61              }else{
62                 WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
63                 OnDrain(args);
64              }
65           }
66           break;
67       } // end switch
68   }
69
70     public void OnFull(WaterLevelEventArgs e){
71       if(Full != null){
72          Full(e);
73       }
74      }
75
76     public void OnEmpty(WaterLevelEventArgs e){
77        if(Empty != null){
78          Empty(e);
79        }
80     }
81
82     public void OnFill(WaterLevelEventArgs e){
83       if(Fill != null){
84          Fill(e);
85        }
86     }
87
88     public void OnDrain(WaterLevelEventArgs e){
89       if(Drain != null){
90          Drain(e);
91       }
92     }
93  }// end WaterLevelClass definition
```

Referring to Example 13.8 — the WaterLevelSensor is a primary component of the water system. Essentially, its purpose is to keep track of a tank's water level. A WaterLevelSensor object functions in one of two modes of operation as defined by the *Mode* enumeration. It can be either a *HighLevelIndicator* or a *LowLevelIndicator*. If it's operating as a HighLevelIndicator, it keeps track of rising water added via the WaterLevelChange() method. If the water level is rising, it fires the *Fill* event. When the water level reaches the set point, it fires the *Full* event.

If a WaterLevelSensor is operating in the LowLevelIndicator mode, it responds to falling water levels by firing the *Drain* event until the water reaches the low set point, at which time it fires the *Empty* event.

Each of the four events — Fill, Full, Drain, and Empty — are of type WaterLevelEventHander delegate. The Pump class, shown in the following example, defines four event handler methods that respond to each of these events.

13.9 Pump.cs

```
1    using System;
2
3    public class Pump {
4
5      private int _pumpingCapacity;
6      private WaterTank _itsTank;
7
8      public int PumpingCapacity {
9        get { return _pumpingCapacity; }
10       set { _pumpingCapacity = value; }
11     }
12
13     public Pump(WaterTank tank, int pumpingCapacity){
14       PumpingCapacity = pumpingCapacity;
15       _itsTank = tank;
16     }
17
18     public void FullTankEventHandler(WaterLevelEventArgs e){
19       Console.WriteLine("FullTankEventHandler: Draining the water tank!");
20       _itsTank.ChangeWaterLevel(-PumpingCapacity);
21     }
22
23     public void EmptyTankEventHandler(WaterLevelEventArgs e){
24       Console.Write("EmptyTankEventHandler: ");
```

 © 2008 Rick Miller — All Rights Reserved

```
25        Console.WriteLine("Water tank has been drained! The water tank contains " +
26                         e.WaterLevel + " gallons!");
27      }
28
29      public void FillTankEventHandler(WaterLevelEventArgs e){
30        Console.Write("FillTankEventHandler: ");
31        Console.WriteLine("The water tank contains " + e.WaterLevel + " gallons!");
32      }
33
34      public void DrainTankEventHandler(WaterLevelEventArgs e){
35        Console.Write("DrainTankEventHandler: ");
36        Console.WriteLine("The water tank contains " + e.WaterLevel + " gallons!");
37        _itsTank.ChangeWaterLevel(-PumpingCapacity);
38
39      }
40   }
```

Referring to Example 13.9 — a Pump object is created with an associated WaterTank object and a pumping capacity. Water added to the tank causes a Fill event to fire. The FillTankEventHandler() method responds by printing the value of the tank's current water level to the console. Note that the current water level is determined by reading the WaterLevelEventArgs.WaterLevel property. When the water level reaches the high level sensor's set point, the sensor fires the Full event that calls the Pump's FullTankEventHandler() method. This starts the automatic draining process by calling the WaterTank.ChangeWaterLevel() method with a negative amount of water equal to the volume of its pumping capacity. This in turn triggers a Drain event in a WaterLevelSensor object, which calls the pump's DrainTankEventHandler() method. This results in yet another call (recursive) to the WaterTank.ChangeWaterLevel() method with a negative amount of water equal to the volume of its pumping capacity. Thus, the recursive calls to the DrainTankEventHandler() method repeat until the low-level indicator reaches its set point.

13.10 WaterTank.cs

```
1    using System;
2    using System.Drawing;
3    using System.Windows.Forms;
4
5    public class WaterTank : Panel {
6
7       // Private fields
8       private WaterLevelSensor _highLevelSensor;
9       private WaterLevelSensor _lowLevelSensor;
10      private Pen _whitePen;
11      private Pen _bluePen;
12      private int _penWidth;
13      private int _currentWaterLevel;
14      private int _lastWaterLevel;
15      private int _tankCapacity;
16      private Point _bottomLeft;
17      private Point _bottomRight;
18      private Pump _itsPump;
19      private Graphics _graphics;
20
21     // Constants
22      private const int UPPER_LEFT_CORNER_X = 100;
23      private const int UPPER_LEFT_CORNER_Y = 100;
24      private const int WIDTH = 100;
25      private const int HEIGHT = 500;
26      private const int TANK_CAPACITY = 10000;
27      private const int PUMP_CAPACITY = 1000;
28      private const int ONE_PIXEL_WIDE = 1;
29      private const int EMPTY = 0;
30
31     public int WaterLevel {
32        get {return _currentWaterLevel; }
33      }
34
35      public int FillRate {
36        get {return _itsPump.PumpingCapacity; }
37      }
38
39      public int HighSetPoint {
40        get { return _highLevelSensor.SetPoint; }
41        set { _highLevelSensor.SetPoint = value; }
42      }
43
44      public int LowSetPoint {
45        get { return _lowLevelSensor.SetPoint; }
46        set { _lowLevelSensor.SetPoint = value; }
47      }
48
```

 © 2008 Rick Miller — All Rights Reserved

```
49      public WaterTank(int x, int y, int width, int height, int tankCapacity, int pumpCapacity){
50        this.InitializeComponents(x, y, width, height, tankCapacity, pumpCapacity);
51      }
52
53      public WaterTank():this(UPPER_LEFT_CORNER_X, UPPER_LEFT_CORNER_Y, WIDTH, HEIGHT, TANK_CAPACITY,
54                              PUMP_CAPACITY){ }
55
56      private void InitializeComponents(int x, int y, int width, int height, int tankCapacity,
57                                          int pumpCapacity){
58
59        this.Bounds = new Rectangle(x, y, width, height);
60        this.BackColor = Color.White;
61        this.BorderStyle = BorderStyle.Fixed3D;
62        _graphics = this.CreateGraphics();
63        _bottomLeft = new Point(0, height);
64        _bottomRight = new Point(width, height);
65        _tankCapacity = tankCapacity;
66        _currentWaterLevel = EMPTY;
67        _itsPump = new Pump(this, pumpCapacity);
68        _penWidth = this.Height/(_tankCapacity/_itsPump.PumpingCapacity);
69        if(_penWidth < 1) _penWidth = 1;
70        _whitePen = new Pen(Color.White, _penWidth);
71        _bluePen = new Pen(Color.Blue, _penWidth);
72        _highLevelSensor = new WaterLevelSensor(tankCapacity - pumpCapacity, EMPTY);
73        _highLevelSensor.SensorMode = WaterLevelSensor.Mode.HighLevelIndicator;
74        _highLevelSensor.Fill += new WaterLevelEventHandler(_itsPump.FillTankEventHandler);
75        _highLevelSensor.Full += new WaterLevelEventHandler(_itsPump.FullTankEventHandler);
76        _lowLevelSensor = new WaterLevelSensor(pumpCapacity, EMPTY);
77        _lowLevelSensor.SensorMode = WaterLevelSensor.Mode.LowLevelIndicator;
78        _lowLevelSensor.Drain += new WaterLevelEventHandler(_itsPump.DrainTankEventHandler);
79        _lowLevelSensor.Empty += new WaterLevelEventHandler(_itsPump.EmptyTankEventHandler);
80      }
81
82      public void ChangeWaterLevel(int amount){
83        _lowLevelSensor.WaterLevelChange(amount);
84        _highLevelSensor.WaterLevelChange(amount);
85        _currentWaterLevel += amount;
86        _lastWaterLevel = _currentWaterLevel;
87        this.ChangeVisualLevel(amount);
88      }
89
90      private void ChangeVisualLevel(int amount){
91        if(amount > 0){
92          _graphics.DrawLine(_bluePen, _bottomLeft, _bottomRight);
93          _bottomLeft.Y -= _penWidth;
94          _bottomRight.Y -= _penWidth;
95
96        }else{
97          _graphics.DrawLine(_whitePen, _bottomLeft, _bottomRight);
98          _bottomLeft.Y += _penWidth;
99          _bottomRight.Y += _penWidth;
100         Delay(30000000);
101       }
102
103     } // end ChangeVisualLevel method
104
105     private void Delay(long ticks){
106       for(long i = 0; i<ticks; i++){
107         ;
108       }
109     }
110   } // end class definition
```

Referring to Example 13.10 — the WaterTank class is an aggregate of a Pump and two WaterLevelSensors. It also provides a visual representation of a water tank by animating the rising and falling water level via blue and white lines drawn on a Panel. Most of the action occurs in three methods: InitializeComponents(), ChangeWaterLevel(), and ChangeVisualLevel(). (**Note:** An attempt is made to keep the visual filling animation in step with the tank's water level, however, when the value of _penWidth reaches 1, the animation gets a little goofy!)

The WaterLevelSensor objects are created in the InitializeComponents() method. One is designated as the _highLevelSensor and the other the _lowLevelSensor. Each sensor's SetPoint is set via its constructor followed by its SensorMode property. Next, the Pump's event handler methods are registered with each sensor's respective events.

Water is added to the tank via the ChangeWaterLevel() method. This in turn makes a call to each sensor's WaterLevelChange() method. The tank's level values are adjusted and finally its visual state is changed with a call to its ChangeVisualLevel() method. The Delay() method is used to slow down the draining animation so you can watch the water level drop.

13.11 WaterSystemApp.cs

© 2008 Rick Miller — All Rights Reserved

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4
5    public class WaterSystemApp : Form {
6
7       private FlowLayoutPanel _panel;
8       private Button _button;
9       private WaterTank _tank;
10
11      public WaterSystemApp(){
12         this.InitializeComponents();
13      }
14
15      public void InitializeComponents(){
16         _tank = new WaterTank();
17         _button = new Button();
18         _button.Text = "Add Water";
19         _button.Click += new EventHandler(this.AddWaterButtonClick);
20         _button.Dock = DockStyle.Bottom;
21         _panel = new FlowLayoutPanel();
22         _panel.SuspendLayout();
23         _panel.FlowDirection = FlowDirection.TopDown;
24         _panel.AutoSize = true;
25         _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
26         _panel.Height = _tank.Height + _button.Height + 75;
27         _panel.Controls.Add(_tank);
28         _panel.Controls.Add(_button);
29         this.SuspendLayout();
30         this.Text = "Water System";
31         this.Height = _panel.Height;
32         this.Width = _tank.Width;
33         this.Controls.Add(_panel);
34         _panel.ResumeLayout();
35         this.ResumeLayout();
36      }
37
38      public void AddWaterButtonClick(object sender, EventArgs e){
39          _tank.ChangeWaterLevel(_tank.FillRate);
40      }
41
42      public static void Main(){
43         Application.Run(new WaterSystemApp());
44      }
45   }// end WaterSystemApp class definition
```

Referring to Example 13.11 — the WaterSystemApp class extends Form and provides the user interface for the water system application. It creates a FlowLayoutPanel and adds to it the WaterTank, which is itself a panel, and a button. Each time the button is clicked, water is added to the tank in an amount equal to the tank's *FillRate* property. The WaterTank.FillRate property is read-only and equals the value of its pump's PumpingCapacity. Figure 13-6 shows the results of running this program. However, you'll learn more from the program by running it and seeing for yourself how the events actually work. Experimenting with different tank dimensions and pumping capacities is left as an exercise.

Naming Conventions

If you'll pause for a moment to consider the previous two custom event examples, you'll notice a few similarities in the names given to certain components and methods. It helps to clarify the purpose of each component or method by adopting the following or similar naming convention.

- Add the suffix "EventArgs" to your event argument class names.
- Add the suffix "EventHandler" to your event-handler delegate names.
- Add the prefix "On" to the event name for the method that fires the event. (*i.e.*, OnFill())
- Add the suffix "Handler" or optionally "ClassName + EventName" to your event handler methods. (*i.e.*, FillTankEventHandler() or AddWaterButtonClick())

© 2008 Rick Miller — All Rights Reserved

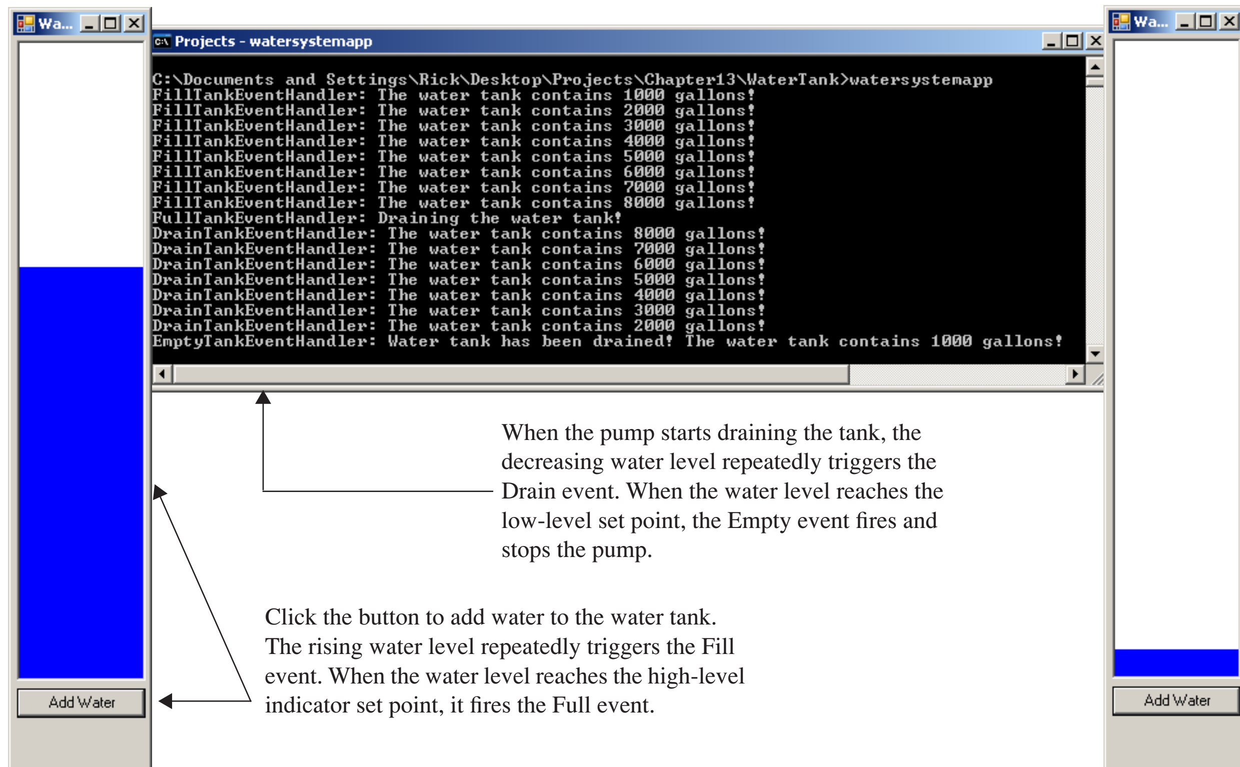


Figure 13-6: Results of Running Example 13.11

Final Thoughts On Extending The EventArgs Class

In the previous two programming examples, I created custom event argument classes by extending the EventArgs class, but this was not strictly necessary, since I didn't use any of the functionality provided by the EventArgs class. In fact, the EventArgs class does nothing except provide a future evolutionary path for the .NET event API by serving as the base class for all the .NET event argument classes.

Summary

You need two logical components to implement the event processing model: 1) an event producer (*publisher*), and 2) an event consumer (*subscriber*). A *delegate* type specifies the authorized method signature for event handler methods. A delegate object maintains a list of event subscribers in the form of references to event handler methods. An event's subscriber list is initially empty until the first subscriber has been added to it with the '+=' operator. Event handler methods must conform to the method signature specified by an event's delegate type.

It helps to clarify the purpose of each component or method if you adopt the following or similar naming convention: add the suffix "EventArgs" to your event argument class name, add the suffix "EventHandler" to your event handler delegate names, add the prefix "On" to the event name for the method that fires the event, and finally, add the suffix "Handler" or optionally "ClassName + EventName" to your event handler methods.

It's not necessary to extend the System.EventArgs class to create custom event argument classes. The EventArgs class does nothing except provide a future evolutionary path for the .NET event API by serving as the base class for all the .NET event argument classes.

 © 2008 Rick Miller — All Rights Reserved

Skill-Building Exercises

1. **Compile and Execute Example Code:** Compile and execute the MinuteTick code given in Examples 13.1 through 13.5.

2. **Compile and Execute Example Code:** Compile and execute the automated water tank system code given in Examples 13.6 through 13.11.

3. **Create UML Sequence Diagram:** Create a UML sequence diagram for the Publisher.CountMinutes() method given in Example 13.3.

4. **Create UML Sequence Diagram:** Create a UML sequence diagram for the WaterTank.ChangeWaterLevel() method given in Example 13.10. You may actually need to create several diagrams to document the call to each WaterLevelSensor's WaterLevelChange() method. Pay particular attention to where the code starts to make recursive calls when the water tank is being drained.

5. **Have Some Fun:** Experiment with the automated water tank system code. Change the value of the Pump's pumping capacity and note the effects of the animation.

6. **API Drill:** Explore the .NET API and find all the delegates. List each delegate and describe its purpose.

Suggested Projects

1. **Oil Tanker Pumping System:** Revisit the oil tanker pumping system project given in Chapter 11, suggested project 5. Add a GUI that displays the tanks and their levels, the valves and their status (open or closed), and the pumps and their status. Utilize events to help monitor tank levels and to automatically start and stop the pumps. This will be a great project to let your imagination run wild!

2. **Refactor Code:** Using your knowledge of inheritance, redesign the WaterLevelSensor class given in Example 13.8 so that there are two separate classes called HighLevelSensor and LowLevelSensor. These two classes should derive from a common base class. Make the necessary modifications to the automated water tank system application.

3. **Modify Code:** Change the MinuteTick application presented in this chapter to respond to and display second ticks. (*i.e.*, every second vs. every minute).

Self-Test Questions

1. What two logical components are required to implement the event processing model?

2. What's the purpose of a delegate type?

3. Which keyword declares a new delegate type?

4. What's the relationship between delegates and events?

5. What's the relationship between delegates and event handler methods?

© 2008 Rick Miller — All Rights Reserved

6. Which operator do you use register an event handler method with an object's event?

7. In what type of data structure does a delegate maintain its subscriber list?

8. In what order does the delegate make calls to its registered event handler methods?

9. What's the value of a delegate object that has no registered event handler methods?

10. Are you required to extend the EventArgs class to create custom event argument classes?

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

Notes

© 2008 Rick Miller — All Rights Reserved

Part IV: Intermediate Concepts

© 2008 Rick Miller — All Rights Reserved

 © 2008 Rick Miller — All Rights Reserved

Chapter 14

Pentax 67 / SMC Takumar 150/2.8 / Ilford Delta 400

Jill & Ryan

Collections

Learning Objectives

- *Describe the purpose of a collection*
- *List and describe the classes, interfaces, and structures contained in the System.Collections, System.Collections.Generic, and System.Collections.ObjectModel namespaces*
- *State the general performance characteristics of arrays, lists, trees, and hash tables*
- *State the general operational characteristics of stacks and queues*
- *Choose a collection based on anticipated usage patterns*
- *Create custom collection classes by extending existing collection classes*
- *State the difference between non-generic and generic collections*
- *State the purpose of a class indexer member*
- *State the purpose of enumerators*
- *Use the foreach statement to iterate over a collection*
- *Utilize non-generic collections in your programs*
- *Utilize generic collections in your programs*

© 2008 Rick Miller — All Rights Reserved

Introduction

When considering all functional assets of the .NET API, none possess the extraordinary potential to save you more time and hassle than its collections framework. Spanning four namespaces: System.Collections, System.Collections.Generic, System.Collections.ObjectModel, and System.Collections.Specialized, the interfaces, classes, and structures belonging to the collections framework provide you with a convenient way to manipulate collections of objects.

This chapter introduces you to the .NET collections framework and shows you how to employ its interfaces, classes, and structures in your programs. Along the way, you will learn about the functional characteristics of arrays, linked lists, hash tables, and red-black trees. Knowledge of these concepts sets the stage for understanding the inner workings of collections framework classes.

From the beginning, the .NET Framework API has undergone continuous evolutionary improvement. Nowhere is this more evident than in the collections framework. Each subsequent release of the .NET Framework introduces improved collections capability.

The differences between the .NET 2.0 collections framework and the previous version are significant. Version 2.0 introduced generics. 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 non-generic versions of the 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 type. Such use of casting renders code hard to read and proves difficult for novice programmers to master.

I begin this chapter 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 then show you how to manipulate collections using non-generic collection classes and interfaces. You will find this treatment handy primarily because you may find yourself faced with maintaining legacy .NET code. I then go on to discuss improvements made to the collections framework introduced with .NET 2.0 and later versions of the collections framework.

Case Study: Building A Dynamic Array

Imagine for a moment that you are working on a 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 14.1 along with a short test program shown in Example 14.2.

14.1 DynamicArray.cs

```
1    using System;
2
3    public class DynamicArray {
4      private Object[] _object_array = null;
5      private int _next_open_element = 0;
6      private int _growth_increment = 10;
7      private const int INITIAL_SIZE = 25;
8
9      public int Count {
10        get { return _next_open_element; }
11     }
12
13     public object this[int index] {
14        get {
15          if((index >= 0) && (index < _object_array.Length)){
16             return _object_array[index];
17         }else return null;
18        }
19        set {
20         if(_next_open_element < _object_array.Length){
21               _object_array[_next_open_element++] = value;
22            }else{
23          GrowArray();
```


© 2008 Rick Miller — All Rights Reserved

```
24          _object_array[_next_open_element++] = value;
25       }
26      }
27     }
28
29     public DynamicArray(int size){
30       _object_array = new Object[size];
31     }
32
33     public DynamicArray():this(INITIAL_SIZE){ }
34
35     public void Add(Object o){
36       if(_next_open_element < _object_array.Length){
37          _object_array[_next_open_element++] = o;
38       }else{
39           GrowArray();
40           _object_array[_next_open_element++] = o;
41        }
42     } // end add() method;
43
44     private void GrowArray(){
45       Object[] temp_array = _object_array;
46       _object_array = new Object[_object_array.Length + _growth_increment];
47       for(int i=0, j=0; i<temp_array.Length; i++){
48         if(temp_array[i] != null){
49           _object_array[j++] = temp_array[i];
50          }
51          _next_open_element = j;
52        }
53        temp_array = null;
54     } // end growArray() method
55   } // end DynamicArray class definition
```

Referring to Example 14.1 — 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 to grow the array, it will expand by 10 elements. In addition to its two constructors, the DynamicArray class has one property named Count, two additional methods named Add() and GrowArray(), and a class indexer member that starts on line 13. An indexer is a member that allows an object to be indexed the same way as an array.

The Add() method inserts an object reference into the next available array element pointed to by the _next_open_element variable. If the array is full, the GrowArray() method is called to grow the array. The GrowArray() method creates a temporary array of objects and copies each element to the temporary array. It then creates a new, larger object array, and copies the elements to it from the temporary array.

The indexer member allows you to access each element of the array. If the index argument falls out of bounds, the indexer returns null. The Count property simply returns the number of elements (references) contained in the array, which is the value of the _next_open_element variable.

Example 14.2 shows the DynamicArray class in action.

14.2 ArrayTestApp.cs

```
1    using System;
2
3    public class ArrayTestApp {
4      public static void Main(){
5        DynamicArray da = new DynamicArray();
6        Console.WriteLine("The array contains " + da.Count + " objects.");
7        da.Add("Ohhh if you loved C# like I love C#!!");
8        Console.WriteLine(da[0].ToString());
9        for(int i = 1; i<26; i++){
10         da.Add(i);
11       }
12       Console.WriteLine("The array contains " + da.Count + " objects.");
13       for(int i=0; i<da.Count; i++){
14         if(da[i] != null){
15          Console.Write(da[i].ToString() + ", ");
16          }
17       }
18       Console.WriteLine();
19     }//end Main() method
20   }// end ArrayTestApp class definition
```

Referring to Example 14.2 — on line 5, an instance of DynamicArray is created using the default constructor. This results in an initial internal array length of 25 elements. Initially, its Count is zero because no references have yet been inserted. On line 7, a string object is added to the array and then printed to the console on line 8. The `for` state-

 © 2008 Rick Miller — All Rights Reserved

ment on line 9 inserts enough integers to test the array's growth capabilities. The `for` statement on line 13 prints all the non-null elements to the console. Figure 14-1 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\DynamicArray>arraytestapp
The array contains 0 objects.
Ohhh if you loved C# like I love C#!!
The array contains 26 objects.
Ohhh if you loved C# like I love C#!!, 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,

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\DynamicArray>
```

Figure 14-1: Results of Testing DynamicArray

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 use it 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 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 IList interface and the ArrayList class, both of which belong to the .NET collections framework. Example 14.3 gives the code.

14.3 ArrayTestApp.cs (Mod 1)

```
1    using System;
2    using System.Collections;
3
4    public class ArrayTestApp {
5      public static void Main(){
6       IList da = new ArrayList();
7       Console.WriteLine("The array contains " + da.Count + " objects.");
8       da.Add("Ohhh if you loved C# like I love C#!!");
9       Console.WriteLine(da[0].ToString());
10      for(int i = 1; i<26; i++){
11         da.Add(i);
12      }
13      Console.WriteLine("The array contains " + da.Count + " objects.");
14      for(int i=0; i<da.Count; i++){
15         if(da[i] != null){
16         Console.Write(da[i].ToString() + ", ");
17         }
18      }
19      Console.WriteLine();
20     }//end Main() method
21   }// end ArrayTestApp class definition
```

Referring to Example 14.3 — I made only three changes to the original ArrayTestApp program: 1) I added another `using` directive on line 2 to provide access to the System.Collections namespace, 2) I changed the `da` reference declared on line 6 from a DynamicArray type to an IList interface type, and 3) also on line 6, I created an instance of ArrayList instead of an instance of DynamicArray.

Figure 14-2 shows the results of running this program. If you compare figures 14-1 and 14-2, you will see that the output produced with an ArrayList is exactly the same as that produced using the DynamicArray. However, the ArrayList class provides much more ready-made functionality.

© 2008 Rick Miller — All Rights Reserved

```
Projects
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\ArrayList>arraytestapp
The array contains 0 objects.
Ohhh if you loved C# like I love C#!!
The array contains 26 objects.
Ohhh if you loved C# like I love C#!!, 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,

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\ArrayList>_
```

Figure 14-2: Results of Running Example 14.3

You might be asking yourself, "Why does this code work?" As it turns out, I gamed the system. The DynamicArray class presented in Example 14.1 just happens to partially implement the IList interface. Later, in Example 14.3, when I changed the type of the da reference from DynamicArray to IList and used the ArrayList collection class, everything worked fine because the ArrayList class implements the IList interface as well. In fact, I could substitute for ArrayList any collection class that implements the IList interface. **Note:** Unfortunately, not many do, as you'll learn when you dive deeper into the Collections namespace.

A Quick Peek At Generics

I can modify Example 14.1 once again to use a generic collection class. Example 14.4 gives the code.

14.4 ArrayTestApp.cs (Mod 2)

```
1    using System;
2    using System.Collections.Generic;
3
4    public class ArrayTestApp {
5      public static void Main(){
6        IList<Object> da = new List<Object>();
7        Console.WriteLine("The array contains " + da.Count + " objects.");
8        da.Add("Ohhh if you loved C# like I love C#!!");
9        Console.WriteLine(da[0].ToString());
10       for(int i = 1; i<26; i++){
11         da.Add(i);
12       }
13       Console.WriteLine("The array contains " + da.Count + " objects.");
14       for(int i=0; i<da.Count; i++){
15         if(da[i] != null){
16          Console.Write(da[i].ToString() + ", ");
17          }
18       }
19       Console.WriteLine();
20     }//end Main() method
21   }// end ArrayTestApp class definition
```

Referring to Example 14.4 — on line 6, I changed the da reference's type to IList<Object> and created an instance of the List<Object> generic collection class. (*i.e.,* List<T> where you substitute for T the type of objects you want the collection to contain.) Again, this code works because the IList<T> generic interface declares the same methods as the non-generic IList interface does. Figure 14-3 shows the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\List>arraytestapp
The array contains 0 objects.
Ohhh if you loved C# like I love C#!!
The array contains 26 objects.
Ohhh if you loved C# like I love C#!!, 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,

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\List>_
```

Figure 14-3: Results of Running Example 14.4

Quick Review

The .NET collections framework can potentially save you a lot of time and hassle. It contains classes, structures, and interfaces designed to make it easy to manipulate collections of objects. The .NET 2.0 framework introduced generic collections and improved performance.

© 2008 Rick Miller — All Rights Reserved

Data Structure Performance Characteristics

In this section, I want to introduce you to the performance characteristics of several different types of foundational data structures. These include the *array*, *linked list*, *hash table*, and *red-black binary tree*. Knowing a little bit about how these data structures work and behave will make it easier for you to select the .NET collection type that's best suited for your particular application.

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 value 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 14-4 through 14-6.

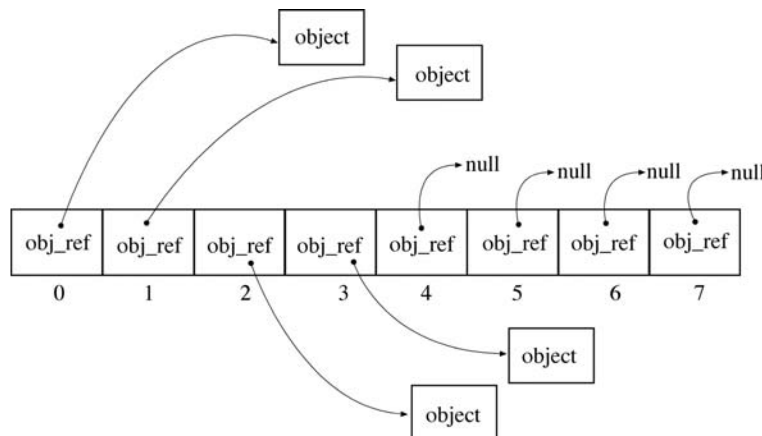


Figure 14-4: Array of Object References Before Insertion

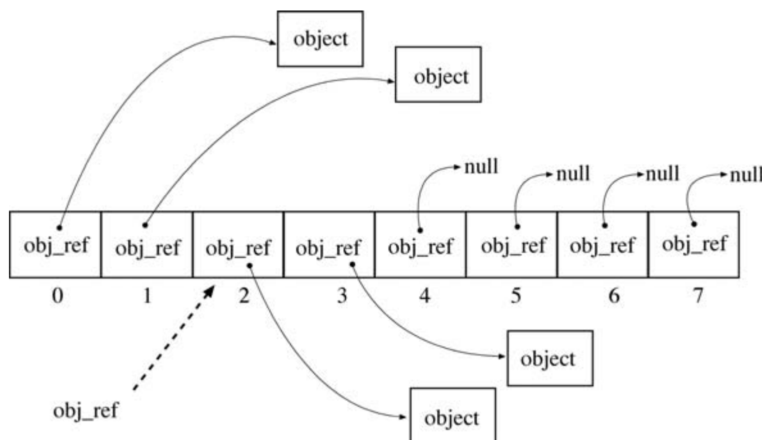


Figure 14-5: New Reference to be Inserted at Array Element 3 (index 2)

© 2008 Rick Miller — All Rights Reserved

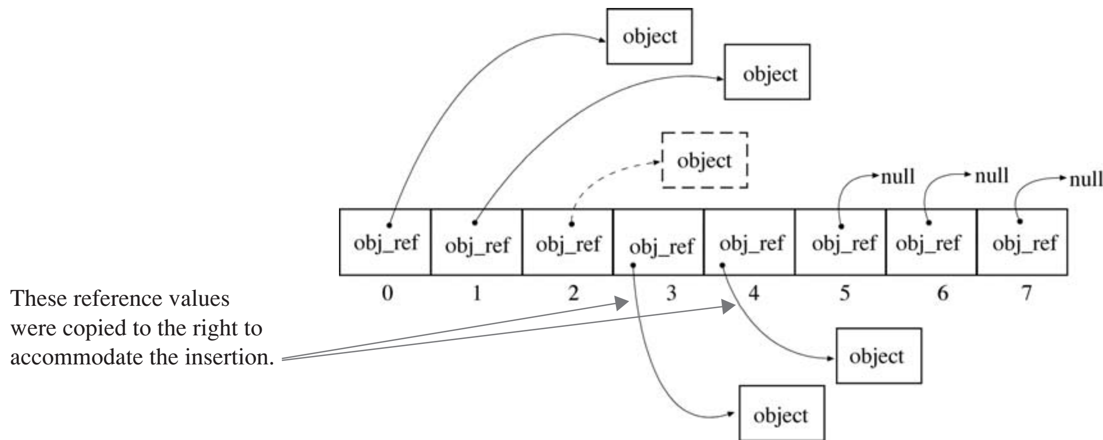


Figure 14-6: Array After New Reference Insertion

Referring to figures 14-4 through 14-6 — 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 that causes 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. (*Refer to Chapter 8 — Arrays*)

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 14-7 illustrates.

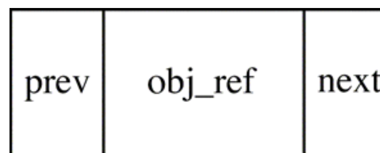


Figure 14-7: Linked List Node Organization

Whereas an array's elements are always located one right after the other in memory, and their memory addresses quickly calculated, a 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 take place fairly quickly because no element shifting is required. Figures 14-8 through 14-11 show the sequence of events for the insertion of a circular linked list node. Referring to figures 14-8 through 14-11 — a linked list contains one or more non-contiguous nodes. A node insertion requires reference rewiring. This entails 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!

© 2008 Rick Miller — All Rights Reserved

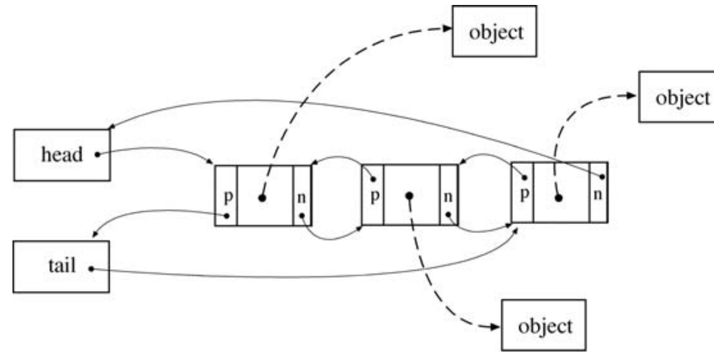


Figure 14-8: Linked List Before New Element Insertion

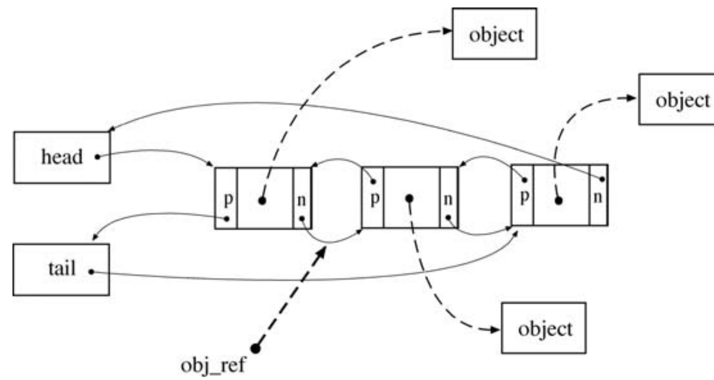


Figure 14-9: New Reference Being Inserted Into Second Element Position

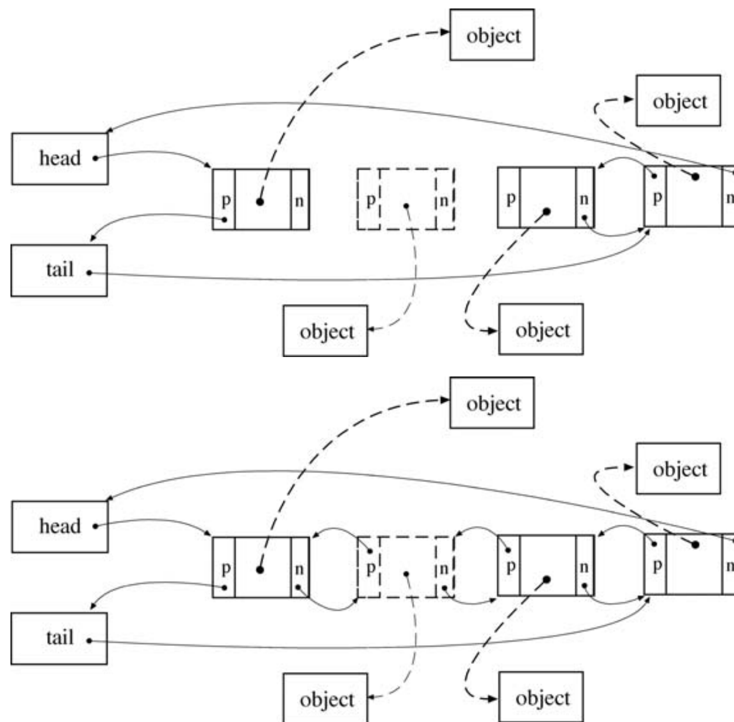


Figure 14-10: References of Previous, New, and Next List Elements must be Manipulated

© 2008 Rick Miller — All Rights Reserved

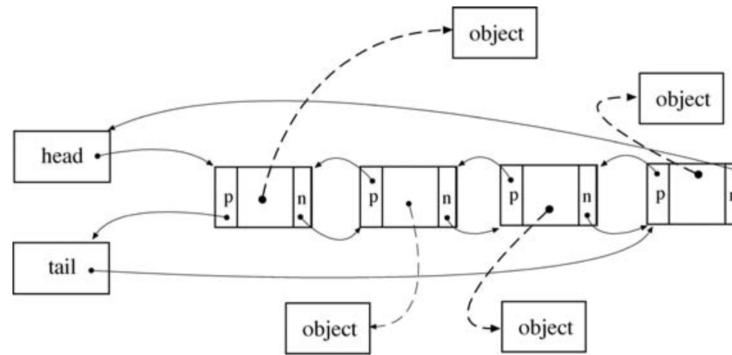


Figure 14-11: Linked List Insertion Complete

Hash Table Performance Characteristics

A hash table is an array whose elements can point to a series of nodes. Structurally, as you'll see, a hash table 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 hash table 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 14-12 illustrates.

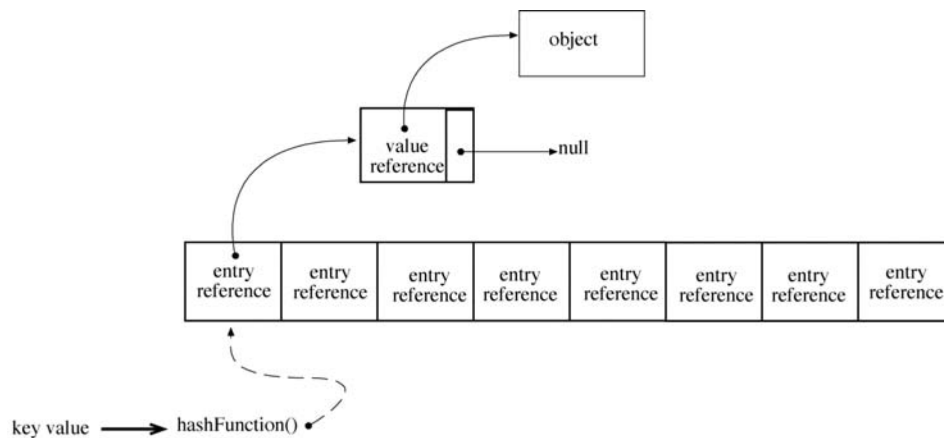


Figure 14-12: A Hash Function Transforms a Key Value into an Array Index

Referring to Figure 14-12 — the purpose of the hash function is to transform the key value into a unique array index value. However, sometimes two unique key values translate to the same index value. When this happens a *collision* is said to have occurred. The problem is resolved by chaining together nodes that share the same hash table index as is shown in Figure 14-13.

The benefits of a hash table 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.

Chained Hash Table vs. Open-Address Hash Table

The hash table discussed above is referred to as a *chained hash table*. Another type of hash table, referred to as an *open-address hash table*, uses a somewhat larger array and replaces the linking mechanism with a *slot probe function* that searches for empty space when the table approaches capacity.

 © 2008 Rick Miller — All Rights Reserved

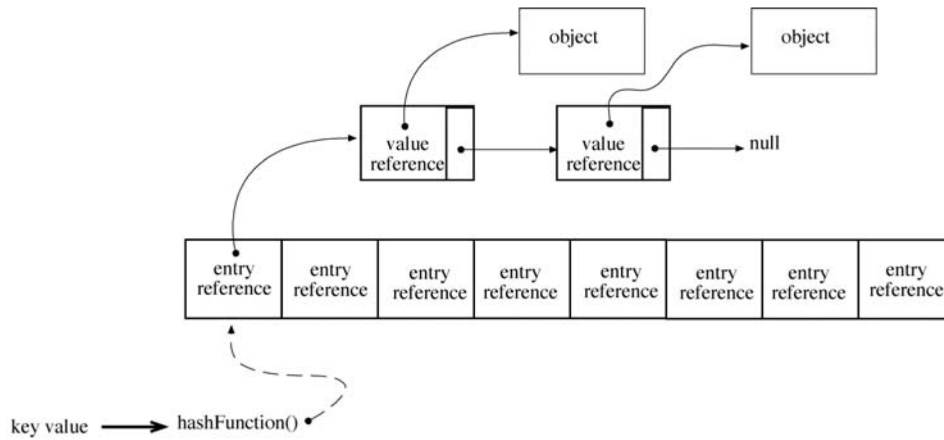


Figure 14-13: Hash Table Collisions are Resolved by Linking Nodes Together

Red-Black Tree Performance Characteristics

A *red-black tree* is a special type of *binary search tree* with a self-balancing characteristic. *Tree nodes* have an additional data element, *color*, that is set to either red or black. The data elements of a red-black tree node are shown in Figure 14-14.

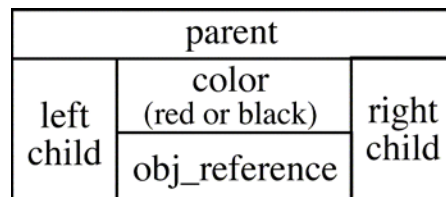


Figure 14-14: Red-Black Tree Node Data Elements

Insertions into a red-black tree are followed by a self-balancing operation. This ensures that all leaf nodes are the same number of black nodes away from the root node. Figure 14-15 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 are shown lightly shaded.)

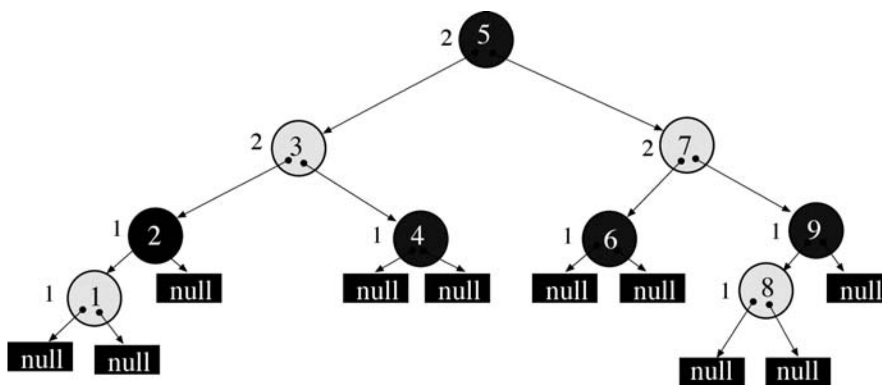


Figure 14-15: Red-Black Tree After Inserting Integer Values 9, 3, 5, 6, 7, 8, 4, 1

Referring to Figure 14-15 — 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 perfor-

 © 2008 Rick Miller — All Rights Reserved

mance 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 incurs 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're looking for. Storing things in a clean room, conversely, takes a little while longer, but when you need something, you can find if fast!

Stacks And Queues

Two additional data structures you'll encounter in the collections API are *stacks* and *queues*. A stack is a data structure that stores objects in a *last-in-first-out* (LIFO) basis. Objects are placed on the stack with a *push* operation and removed from the stack with a *pop* operation. A stack operates like a plate dispenser, where you put in a stack of plates and take plates off the stack one at a time. The last plate inserted into the plate dispenser is the first plate dispensed when someone needs a plate. Figure 14-16 shows the state of a stack after several pushes and pops.

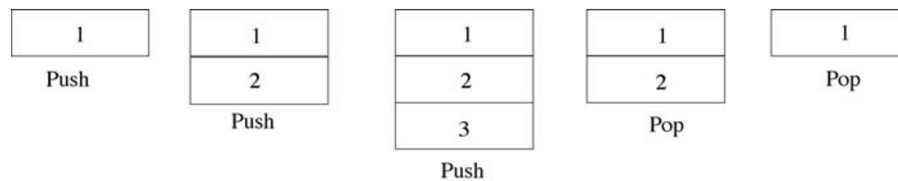


Figure 14-16: A Stack After Several Push and Pop Operations

A *queue* is a data structure that stores objects in a *first-in-first-out* (FIFO) basis. A queue operates like a line of people waiting for some type of service; the first person in line is the first person to be served. People arriving in line must wait for service until they reach the front of the line. Objects are added to a queue with an *enqueue* operation and removed with a *dequeue* operation. Figure 14-17 shows the state of a queue after several enqueues and dequeues.

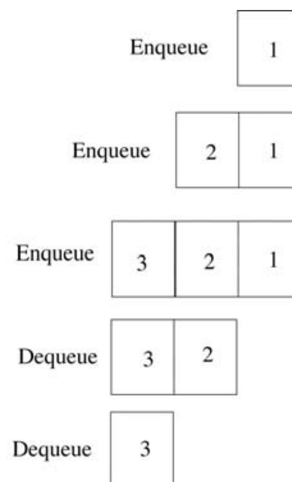


Figure 14-17: A Queue After Several Enqueue and Dequeue Operations

Quick Review

An *array* is a contiguous allocation of objects in memory. An array-based collection offers quick element access but slow element insertion, especially if the collection's underlying array must be resized and its contents shifted to accommodate the insertion.

A *linked list* consists of individual *nodes* linked to each other via references. To traverse a linked list, you must start at the beginning, or the end (head or tail) and follow each element to the next. Linked list-based collections can

© 2008 Rick Miller — All Rights Reserved

conserve memory space because memory need only be allocated on each object insertion. Insertions into linked list-based collections are relatively quick but element access is relatively slow due to the need to traverse the list.

A *chained hash table* is a cross between an array and a linked list and allows element insertion with key/value pairs. A *hash function* performed on the key determines the value's location in the hash table. A collision is said to occur when two keys produce the same hash code. When this happens, the values are chained together in a linked list-like structure. A hash function that produces a uniform distribution over all the keys is a critical feature of a hash table.

A red-black tree is a self-balancing binary tree. Insertions into a red-black tree take some time because of element ordering and balancing operations. Element access times for a red-black tree-based collection is fairly quick.

Navigating The .NET Collections API

To the uninitiated, the .NET collections API presents a bewildering assortment of interfaces, classes, and structures spread over four namespaces. In this section, I provide an overview of some of the things you'll find in each namespace. Afterward, I present a different type of organization that I believe you'll find more helpful.

One thing I will not do in this section is discuss every interface, class, or structure found in these namespaces. If I did that, you would fall asleep quick and kill yourself as your head slammed against the desk on its way down! Instead, I will only highlight the most important aspects of each namespace with an eye towards saving you time and frustration.

One maddening aspect of the .NET collections framework is in the way Microsoft chose to name their collection classes. For example, collection classes that contain the word List do not necessarily implement the IList or IList<T> interfaces. This means you can't substitute a LinkedList for an ArrayList without breaking your code. (In this regard I believe the Java collections framework is much more robust and logically organized.)

In concert with this section you should explore the collections API and see for yourself what lies within each namespace.

System.Collections

The System.Collections namespace contains non-generic versions of collection interfaces, classes, and structures. The contents of the System.Collections namespace represent the "old-school" way of collections programming. By this I mean that the collections defined here store only object references. You can insert any type of object into a collection like an ArrayList, Stack, etc., but, when you access an element in the collection and want to perform an operation on it specific to a particular type, you must first cast the object to a type that supports the operation. (By "performing an operation" I mean accessing an object member declared by its interface or class type.)

I recommend avoiding the System.Collections namespace altogether in favor of the generic versions of its members found in the System.Collections.Generic namespace. In most cases, you'll be trading cumbersome "old-school" style programming for more elegant code and improved performance offered by the newer collection classes.

System.Collections.Generic

.NET 2.0 brought with it generics and the collection classes found in the System.Collections.Generic namespace. In addition to providing generic versions of the "old-school" collections contained in the System.Collections namespace, the System.Collections.Generic namespace added several new collection types, one of them being LinkedList<T>.

Several collection classes within the System.Collections.Generic namespace can be used off-the-shelf, so to speak, to store and manipulate strongly-typed collections of objects. These include List<T>, LinkedList<T>, Queue<T>, Stack<T>.

Other classes such as Dictionary<TKey, TValue>, SortedDictionary<TKey, TValue>, and SortedList<TKey, TValue> store objects (values) in the collection based on the hash values of keys. Special rules must be followed when implementing a key class. These rules specify the types of interfaces a key class must implement in order to perform equality comparisons. They also offer suggestions regarding the performance of hashing functions to opti-

 © 2008 Rick Miller — All Rights Reserved

mize insertion and retrieval. You can find these specialized instructions in the **Remarks** section of a collection class's API documentation page. I cover this topic in greater detail in Chapter 22 — Well Behaved Objects.

System.Collections.ObjectModel

The System.Collections.ObjectModel namespace contains classes that are meant to be used as the base classes for custom, user-defined collections. For example, if you want to create a specialized collection, you can extend the Collection<T> class. This namespace also includes the KeyedCollection<TKey, TItem>, ObservableCollection<T>, ReadOnlyCollection<T>, and ReadOnlyObservableCollection<T> classes.

The KeyedCollection<TKey, TItem> is an abstract class and is a cross between an IList and an IDictionary-based collection in that it is an indexed list of items. Each item in the list can also be accessed with an associated key. Collection elements are not key/value pairs as is the case in a Dictionary, rather, the element is the value and the key is extracted from the value upon insertion. The KeyedCollection<TKey, TItem> class must be extended and you must override its GetKeyForItem() method to properly extract keys from the items you insert into the collection.

The ObservableCollection<T> collection provides an event named CollectionChanged that you can register event handlers with to perform special processing when items are added or removed, or the collection is refreshed.

The ReadOnlyCollection<T> and ReadOnlyObservableCollection<T> classes implement read-only versions of the Collection<T> and ObservableCollection<T> classes.

System.Collections.Specialized

As its name implies, the System.Collections.Specialized namespace contains interfaces, classes, and structures that help you manage specialized types of collections. Some of these include the BitVector32 structure, the ListDictionary, which is a Dictionary implemented as a singly linked list intended for storing ten items or less, StringCollection, which is a collection of strings, and StringDictionary, which is a Dictionary whose key/value pairs are strongly typed to strings rather than objects.

Mapping Non-Generic To Generic Collections

In some cases, the System.Collection.Generic and System.Collections.ObjectModel namespaces provide a corresponding replacement for a collection class in the System.Collections namespace. But sometimes they do not. Table 14-1 lists the non-generic collection classes and their generic replacements, if any, and the underlying data structure implementation.

Non-Generic	Generic	Underlying Data Structure
ArrayList	List<T>	Array
BitArray	*No generic equivalent*	Array
CollectionBase	Collection<T>	Array
DictionaryBase	KeyedCollection<TKey, TItem>	Hash Table & Array
HashTable	Dictionary<TKey, TValue>	Hash Table
Queue	Queue<T>	Array
ReadOnlyCollectionBase	ReadOnlyCollection<T>	Array
SortedList	SortedList<TKey, TValue>	Red-Black Tree
Stack	Stack<T>	Array
No Non-Generic Equivalent	LinkedList<T>	Doubly Linked List

Table 14-1: Mapping Non-Generic Collections to Their Generic Counterparts

© 2008 Rick Miller — All Rights Reserved

Non-Generic	Generic	Underlying Data Structure
No Non-Generic Equivalent	SortedDictionary<TKey, TValue>	Red-Black Tree
No Non-Generic Equivalent	SynchronizedCollection<T> †	Array
No Non-Generic Equivalent	SynchonizedKeyedCollection<TKey, TItem> †	Hash Table & Array
No Non-Generic Equivalent	SynchronizedReadOnlyCollection<T> †	Array
† Provides thread-safe operation		

Table 14-1: Mapping Non-Generic Collections to Their Generic Counterparts

Quick Review

"Old-school" style .NET collections classes store only object references and require casting when elements are retrieved. You should favor the use of generic collections as they offer strong element typing on insertion and retrieval and improved performance. The classes found in the System.Collections.ObjectModel namespace can serve as the basis for user-defined custom collections. The System.Collections.Specialized namespace contains classes and structures you will find helpful to manage unique collections.

Using Non-Generic Collection Classes - Pre .NET 2.0

In this section, I demonstrate the use of a non-generic collection class. Whereas earlier I used an ArrayList class to store various types of objects like strings and integers, here I use an ArrayList to store objects of type Person. I want to show you how to do two things in particular: 1) cast objects retrieved from a collection to a specified type, and 2) subclass ArrayList and override its methods to provide a strongly-typed collection. You'll find this section helpful if you're tasked with maintaining legacy .NET code based on 1.0 collection classes.

Example 14.5 gives the code for the Person class.

14.5 Person.cs

```
1    using System;
2
3    public class Person {
4
5      //enumeration
6      public enum Sex {MALE, FEMALE};
7
8      // private instance fields
9      private String   _firstName;
10     private String   _middleName;
11     private String   _lastName;
12     private Sex      _gender;
13     private DateTime _birthday;
14
15
16     //private default constructor
17     private Person(){}
18
19     public Person(String firstName, String middleName, String lastName,
20                   Sex gender, DateTime birthday){
21        FirstName = firstName;
22        MiddleName = middleName;
23        LastName = lastName;
24        Gender = gender;
25        BirthDay = birthday;
26     }
27
28     // public properties
29     public String FirstName {
30       get { return _firstName; }
31       set { _firstName = value; }
32     }
33
34     public String MiddleName {
```

 © 2008 Rick Miller — All Rights Reserved

```
35        get { return _middleName; }
36        set { _middleName = value; }
37      }
38
39      public String LastName {
40        get { return _lastName; }
41        set { _lastName = value; }
42      }
43
44      public Sex Gender {
45        get { return _gender; }
46        set { _gender = value; }
47      }
48
49      public DateTime BirthDay {
50        get { return _birthday; }
51        set { _birthday = value; }
52      }
53
54      public int Age {
55        get {
56          int years = DateTime.Now.Year - _birthday.Year;
57          int adjustment = 0;
58            if(DateTime.Now.Month < _birthday.Month){
59              adjustment = 1;
60            }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
61                    adjustment = 1;
62                  }
63          return years - adjustment;
64        }
65      }
66
67      public String FullName {
68        get { return FirstName + " " + MiddleName + " " + LastName; }
69      }
70
71      public String FullNameAndAge {
72        get { return FullName + " " + Age; }
73      }
74
75      public override String ToString(){
76        return FullName + " is a " + Gender + " who is " + Age + " years old.";
77      }
78    } // end Person class
```

Objects In – Objects Out: Casting 101

"Old school" collections programming is characterized by the need to cast objects to an appropriate type when they are retrieved from a collection. Casting is required if you intend to perform an operation on an object other than those defined by the System.Object class. Example 14.6 gives the code for a short program that stores Person objects in an ArrayList.

14.6 MainApp.cs

```
1     using System;
2     using System.Collections;
3
4     public class MainApp {
5        public static void Main(){
6          ArrayList surrealists = new ArrayList();
7
8          Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
9          Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
10         Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
11         Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
12         Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
13         Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
14                                new DateTime(1887, 07, 28));
15
16         surrealists.Add(p1);
17         surrealists.Add(p2);
18         surrealists.Add(p3);
19         surrealists.Add(p4);
20         surrealists.Add(p5);
21         surrealists.Add(p6);
22
23         foreach(object o in surrealists){
24           Console.WriteLine(o.ToString());
25         }
```

 © 2008 Rick Miller — All Rights Reserved

```
26
27        Console.WriteLine("-----------------------------------------------------");
28
29        foreach(Person p in surrealists){
30          Console.WriteLine(p.FirstName);
31        }
32
33        Console.WriteLine("-----------------------------------------------------");
34
35        for(int i = 0; i<surrealists.Count; i++){
36          Console.WriteLine(((Person)surrealists[i]).FirstName + " " + ((Person)surrealists[i]).LastName);
37        }
38
39      } // end Main()
40    } // end MainApp class definition
```

Referring to Example 14.6 — an ArrayList reference named surrealists is initialized on line 6 followed by the creation of six Person references on lines 8 through 13. Next, each Person reference is inserted into the collection using the ArrayList.Add() method.

The `foreach` statement on line 23 demonstrates how objects can be retrieved from the collection without the need for casting as long as you only intend to call methods defined by the System.Object class. In this case, the Person class overrides the ToString() method.

The `foreach` statement on line 29 demonstrates how to extract a particular type of object from the collection using an *implicit cast*. This works because all the objects in the ArrayList collection are indeed of type Person.

The `for` statement on line 35 iterates over the surrealists collection using array indexing. Each object in the collection must be explicitly cast to type Person before accessing its FirstName and LastName properties.

Figure 14-18 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\ArrayList_Person>mainapp
Rick  Miller is a MALE who is 46 years old.
Max  Ernst is a MALE who is 116 years old.
Andre  Breton is a MALE who is 111 years old.
Roland  Penrose is a MALE who is 107 years old.
Lee  Miller is a FEMALE who is 100 years old.
Henri-Robert-Marcel  Duchamp is a MALE who is 120 years old.
-----------------------------------------------------
Rick
Max
Andre
Roland
Lee
Henri-Robert-Marcel
-----------------------------------------------------
Rick Miller
Max Ernst
Andre Breton
Roland Penrose
Lee Miller
Henri-Robert-Marcel Duchamp

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\ArrayList_Person>
```

Figure 14-18: Results of Running Example 14.6

Extending ArrayList To Create A Strongly-Typed Collection

You can avoid the need to cast by creating a custom collection. In this section, I show you how to extend the ArrayList class to create a custom collection that stores Person objects. Example 14.7 gives the code for the custom collection named PersonArrayList.

14.7 PersonArrayList.cs

```
1     using System;
2     using System.Collections;
3
4     public class PersonArrayList : ArrayList {
5
6       public PersonArrayList():base(){}
7
8       public new Person this[int index]{
9         get { return (Person) base[index];}
10
11        set { base[index] = (Person) value; }
12      }
13
14      public override int Add(object o){
```

 © 2008 Rick Miller — All Rights Reserved

```
15        return base.Add((Person)o);
16      }
17   }// end PersonArrayList class definition
```

Referring to Example 14.7 — the PersonArrayList class extends ArrayList, overrides its Add() method, and provides a new implementation for its indexer. The PersonArrayList.Add() method accepts an object and casts it to Person before inserting it into the collection using the ArrayList.Add() method. The indexer casts incoming objects (*value*) to type Person before inserting them into the collection. It casts retrieved objects to type Person before their return from the indexer call. The indexer cannot simply be overridden in this case because an overridden method must return the same type as the base class method it overrides. Thus, a new indexer is declared with the help of the `new` keyword.

Example 14.8 shows this custom collection in action.

14.8 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4       public static void Main(){
5          PersonArrayList surrealists = new PersonArrayList();
6
7          Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
8          Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
9          Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
10         Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
11         Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
12         Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
13                                 new DateTime(1887, 07, 28));
14
15         surrealists.Add(p1);
16         surrealists.Add(p2);
17         surrealists.Add(p3);
18         surrealists.Add(p4);
19         surrealists.Add(p5);
20         surrealists.Add(p6);
21
22         for(int i=0; i<surrealists.Count; i++){
23           Console.WriteLine(surrealists[i].FullNameAndAge);
24         }
25
26         surrealists.Remove(p1);
27         Console.WriteLine("---------------------------------------");
28
29         for(int i=0; i<surrealists.Count; i++){
30           Console.WriteLine(surrealists[i].FullNameAndAge);
31         }
32
33      } // end Main()
34   } // end MainApp
```

Referring to Example 14.8 — the important point to note here is there is no need to cast when using the PersonArrayList indexer in the bodies of the `for` statements that begin on lines 22 and 29. On line 26 the ArrayList.Remove() method is used to remove the object referenced by p1 from the collection. Figure 14-19 gives the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Extended_ArrayList>mainapp
Rick  Miller 46
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
---------------------------------------
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Extended_ArrayList>_
```

Figure 14-19: Results of Running Example 14.8

© 2008 Rick Miller — All Rights Reserved

Using Generic Collection Classes – .NET 2.0 and Beyond

The release of the .NET 2.0 framework brought with it generics and generic collections. In this section I show you how generic collection classes can help you write cleaner code and eliminate the need to cast objects upon their retrieval from a collection. I start by showing you how to use the List<T> collection which, as you saw in Table 14-1, is the generic replacement for the ArrayList class. I will also show you how to implement a custom KeyedCollection class.

List<T>: Look Ma, No More Casting!

The List<T> generic collection class is the direct replacement for the ArrayList collection class. It's easy to use the List<T> generic collection. The T that appears between the left and right angle brackets represents the replacement type. You can think of the T as acting like a placeholder for a field in a mail-merge document. Anywhere the T appears, the type that's actually substituted for T will appear in its place when the collection object is created. For example, if you examine the List<T> class's MSDN documentation, you'll find that its Add() method signature is declared to be:

```
public void Add(T item)
```

The parameter named *item* is of type T. So, if you want to create a List collection object that stores Person objects you would do the following:

```
List<Person> person_list = new List<Person>();
```

The compiler substitutes the type Person everywhere in the List class the symbol T appears. Thus, its Add() method now accepts only Person-type objects. Let's see this in action. Example 14.9 gives the code for a program that uses the List<T> collection class to store Person objects.

14.9 MainApp.cs (List<T> version)

```
1    using System;
2    using System.Collections.Generic;
3
4    public class MainApp {
5       public static void Main(){
6          List<Person> surrealists = new List<Person>();
7
8          Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
9          Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
10         Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
11         Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
12         Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
13         Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
14                                new DateTime(1887, 07, 28));
15
16         surrealists.Add(p1);
17         surrealists.Add(p2);
18         surrealists.Add(p3);
19         surrealists.Add(p4);
20         surrealists.Add(p5);
21         surrealists.Add(p6);
22
23         for(int i=0; i<surrealists.Count; i++){
24            Console.WriteLine(surrealists[i].FullNameAndAge);
25         }
26
27         surrealists.Remove(p1);
28         Console.WriteLine("---------------------------------------");
29
30         for(int i=0; i<surrealists.Count; i++){
31            Console.WriteLine(surrealists[i].FullNameAndAge);
32         }
33
34      } // end Main()
35   } // end MainApp
```

Referring to Example 14.9 — note the only differences between this example and the code shown in Example 14.8 is the addition of the *using* directive on line 2 to include the System.Collections.Generic namespace. Also, I changed the type of the *surrealists* reference on line 6 from PersonArrayList to List<T>. Figure 14.20 shows the results of running this program.

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\List_T_Person>mainapp
Rick  Miller 46
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
---------------------------------------
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\List_T_Person>_
```

Figure 14-20: Results of Running Example 14.9

Implementing KeyedCollection<TKey, TItem>

You will find the KeyedCollection<TKey, TItem> in the System.Collections.ObjectModel namespace. In this section, I want to show you how to use this particular class because you can't just use it directly like you can List<T>. It is an abstract class and is meant to be used as the basis for a custom collection. It contains one abstract method, GetKeyForItem(), that you must override in order to properly extract the key from the item being inserted into the collection.

To briefly review, the KeyedCollection class functions like a cross between a list and a dictionary. When an item is inserted into the collection, it gets put into an array, just like items inserted into List<T>. When the item is inserted, a key is extracted from it and inserted into an internal dictionary collection. By default, the internal dictionary is created upon the insertion of the first item into the collection, however, by using another version of the KeyedCollection constructor, you can specify the number of items the collection must contain before the internal dictionary is created. This may increase search performance when the collection contains a small number of items.

You can access individual elements in a KeyedCollection in two ways: 1) by array indexing using an integer like you would normally do in an array or a collection that implements the IList<T> interface, or 2) by using an indexer that takes a key as an argument.

Example 14.10 gives the code for a custom KeyedCollection class named PersonKeyedCollection.

14.10 PersonKeyedCollection.cs

```
1    using System;
2    using System.Collections.ObjectModel;
3
4    public class PersonKeyedCollection : KeyedCollection<String, Person> {
5
6       public PersonKeyedCollection():base(){ }
7
8       protected override String GetKeyForItem(Person person){
9          return person.BirthDay.ToString();
10      }
11   }
```

Referring to Example 14.10 — you might be surprised at how little coding it takes to implement the custom KeyedCollection. Note the `using` directive on line 2 specifies the System.Collections.ObjectModel namespace. Also note on line 4 how the class is declared by specifying the actual types of the keys and items. In this example, keys are of type String and items are of type Person.

The overridden GetKeyForItem() method appears on line 8. It simply returns the string representation of a Person object's Birthday property. The GetKeyForItem() method is protected and as such is only used internally by the KeyedCollection class when items are inserted into the collection.

Example 14.11 demonstrates how to use a KeyedCollection.

14.11 KeyedCollectionDemo.cs

```
1    using System;
2
3    public class KeyedCollectionDemo {
4       public static void Main(){
5          PersonKeyedCollection surrealists = new PersonKeyedCollection();
6
7          Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
8          Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
```

 © 2008 Rick Miller — All Rights Reserved

```
9         Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
10        Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
11        Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
12        Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
13                               new DateTime(1887, 07, 28));
14
15        surrealists.Add(p1);
16        surrealists.Add(p2);
17        surrealists.Add(p3);
18        surrealists.Add(p4);
19        surrealists.Add(p5);
20        surrealists.Add(p6);
21
22        for(int i=0; i<surrealists.Count; i++){
23          Console.WriteLine(surrealists[i].FullNameAndAge);
24        }
25
26        surrealists.Remove(p1);
27        Console.WriteLine("---------------------------------------");
28
29        for(int i=0; i<surrealists.Count; i++){
30          Console.WriteLine(surrealists[i].FullNameAndAge);
31        }
32
33        Console.WriteLine("---------------------------------------");
34        Console.WriteLine(surrealists[(new DateTime(1900, 10, 14)).ToString()]);
35        Console.WriteLine(surrealists[(new DateTime(1907, 04, 23)).ToString()]);
36
37      } // end Main()
38    } // end KeyedCollectionDemo class
```

Referring to Example 14.11 — this codes looks a whole lot like that given in Example 14.9 with a few notable exceptions. The surrealists reference type is changed from List<T> to PersonKeyedCollection. Person objects are created and inserted into the collection using the Add() method as usual. Individual collection elements are accessed via an indexer in the fashion of a list.

Lines 34 and 35 demonstrate the use of the overloaded indexer to find elements by their key. In this case, we can find a particular surrealist by his or her birthday string.

Figure 14.21 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\KeyedCollection>keyedcollectiondemo
Rick  Miller 46
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
---------------------------------------
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
---------------------------------------
Roland  Penrose is a MALE who is 107 years old.
Lee  Miller is a FEMALE who is 100 years old.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\KeyedCollection>
```

Figure 14-21: Results of Running Example 14.11

Quick Review

The List<T> generic collection class is the direct replacement for the ArrayList collection class. It's easy to use the List<T> generic collection. The T that appears between the left and right angle brackets represents the replacement type. You can think of the T as acting like a placeholder for a field in a mail-merge document. Anywhere the T appears, the type that's actually substituted for T will appear in its place when the collection object is created.

You can find the KeyedCollection<TKey, TItem> in the System.Collections.ObjectModel namespace. It is an abstract class and is meant to be used as the basis for a custom collection. It contains one abstract method, GetKeyForItem(), that you must override to properly extract the key from the item being inserted into the collection.

© 2008 Rick Miller — All Rights Reserved

Special Operations On Collections

Collections exist to make it easier for you to manipulate the objects they contain. In this section, I show you how to sort a list using several different versions of its overloaded Sort() method. I then show you how to extract the contents of a collection into an array.

Sorting A List

The List<T> collection provides a Sort() method that makes sorting its contents easy. However, before you call the Sort() method, you need to be aware of what types of objects the list contains. You must do this because not all objects are naturally sortable.

To make an object sortable, its class or structure type must implement the IComparable<T> interface. If you don't own the source code for a particular class or structure and wish to sort them using List<T>'s Sort() method, then you can implement a Comparer<T> object that defines how such objects are to be compared with each other. I explain how to do this in the following sections.

Implementing System.IComparable<T>

Fundamental data types provided by the .NET Framework implement the System.IComparable or System.IComparable<T> interface and are therefore sortable. When you create a user-defined class or structure, you must ask yourself, as part of the design process, "Will objects of this type be compared with each other or with other types of objects?" If the answer is yes, then that class or structure should implement the IComparable<T> interface.

The IComparable<T> interface declares one method, CompareTo(). This method is automatically called during collection or array sort operations. If you want your user-defined types to sort correctly, you must provide an implementation for the CompareTo() method. How one object is compared against another is entirely up to you since you're the programmer. It's fun having all the power!

Example 14.12 shows the Person class modified to implement the IComparable<T> interface.

14.12 Person.cs (implementing IComparable<T>)

```
1    using System;
2
3    public class Person : IComparable<Person> {
4
5      //enumeration
6      public enum Sex {MALE, FEMALE};
7
8      // private instance fields
9      private String   _firstName;
10     private String   _middleName;
11     private String   _lastName;
12     private Sex      _gender;
13     private DateTime _birthday;
14
15
16     //private default constructor
17     private Person(){}
18
19     public Person(String firstName, String middleName, String lastName,
20                   Sex gender, DateTime birthday){
21        FirstName = firstName;
22        MiddleName = middleName;
23        LastName = lastName;
24        Gender = gender;
25        BirthDay = birthday;
26     }
27
28     // public properties
29     public String FirstName {
30       get { return _firstName; }
31       set { _firstName = value; }
32     }
33
34     public String MiddleName {
35       get { return _middleName; }
36       set { _middleName = value; }
```

 © 2008 Rick Miller — All Rights Reserved

```
37     }
38
39     public String LastName {
40       get { return _lastName; }
41       set { _lastName = value; }
42     }
43
44     public Sex Gender {
45       get { return _gender; }
46       set { _gender = value; }
47     }
48
49     public DateTime BirthDay {
50       get { return _birthday; }
51       set { _birthday = value; }
52     }
53
54      public int Age {
55         get {
56            int years = DateTime.Now.Year - _birthday.Year;
57            int adjustment = 0;
58            if((DateTime.Now.Month <= _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
59               adjustment = 1;
60            }
61            return years - adjustment;
62       }
63      }
64
65     public String FullName {
66       get { return FirstName + " " + MiddleName + " " + LastName; }
67     }
68
69     public String FullNameAndAge {
70       get { return FullName + " " + Age; }
71     }
72
73     public override String ToString(){
74       return FullName + " is a " + Gender + " who is " + Age + " years old.";
75     }
76
77     public int CompareTo(Person other){
78        return this.BirthDay.CompareTo(other.BirthDay);
79      }
80
81   } // end Person class
```

Referring to Example 14.12 — the Person class now implements the IComparable<T> interface. The CompareTo() method, starting on line 77, defines how one person object is compared with another. In this case, I am comparing their BirthDay properties. Note that since the DateTime structure implements the IComparable<T> interface (*i.e.*, IComparable<DateTime>) one simply needs to call its version of the CompareTo() method to make the required comparison. But what's getting returned? Good question. I answer it in the next section.

Rules For Implementing The CompareTo(T other) Method

The rules for implementing the CompareTo(T other) method are laid out in Table 14-2.

Return Value	Returned When...
Less than Zero (-1)	This object is less than the *other* parameter
Zero (0)	This object is equal to the *other* parameter
Greater than Zero (1)	This object is greater than the *other* parameter, or, the *other* parameter is null

Table 14-2: Rules For Implementing IComparable<T>.CompareTo(T other) Method

What property, exactly, you compare between objects is strictly dictated by the program's design. In the case of the Person class, I chose to compare BirthDays. In the next section, I'll show you how you would compare LastNames.

© 2008 Rick Miller — All Rights Reserved

Now that the Person class implements the IComparable<T> interface, person objects can be compared against each other. Example 14.13 shows the List<T>.Sort() method in action.

14.13 SortingListDemo.cs

```
1    using System;
2    using System.Collections.Generic;
3
4    public class SortingListDemo {
5      public static void Main(){
6        List<Person> surrealists = new List<Person>();
7
8        Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
9        Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
10       Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
11       Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
12       Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
13       Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
14                              new DateTime(1887, 07, 28));
15
16       surrealists.Add(p1);
17       surrealists.Add(p2);
18       surrealists.Add(p3);
19       surrealists.Add(p4);
20       surrealists.Add(p5);
21       surrealists.Add(p6);
22
23       for(int i=0; i<surrealists.Count; i++){
24         Console.WriteLine(surrealists[i].FullNameAndAge);
25       }
26
27       surrealists.Sort();
28       Console.WriteLine("----------------------------------------");
29
30       for(int i=0; i<surrealists.Count; i++){
31         Console.WriteLine(surrealists[i].FullNameAndAge);
32       }
33
34     } // end Main()
35   } // end SortingListDemo
```

Referring to Example 14.13 — the Sort() method is called on line 27. Figure 14-22 shows the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Sorting_List>sortinglistdemo
Rick  Miller 46
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
----------------------------------------
Henri-Robert-Marcel  Duchamp 120
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Rick  Miller 46

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Sorting_List>
```

Figure 14-22: Results of Running Example 14.13

Extending Comparer<T>

What if you don't own the source code to the objects you want to compare? No problem. Simply implement a custom comparer object by extending the System.Collections.Generic.Comparer<T> class and providing an implementation for its Compare(T x, T y) method. Example 14.14 shows how a custom comparer might look. This particular example compares two Person objects.

14.14 PersonComparer.cs

```
1    using System.Collections.Generic;
2
3    public class PersonComparer : Comparer<Person> {
4
5      /**********************************
6         Return -1 if p1 < p2 or p1 == null
7         Return  0 if p1 == p2
```


© 2008 Rick Miller — All Rights Reserved

```
8          Return +1 if p1 > p2 or p2 == null
9       ************************************/
10      public override int Compare(Person p1, Person p2){
11        if(p1 == null) return -1;
12        if(p2 == null) return  1;
13
14        return p1.LastName.CompareTo(p2.LastName);
15      }
16    }
```

Referring to Example 14.14 — the PersonComparer class extends Comparer<T> (i.e., Comparer<Person>) and provides an overriding implementation for its Compare(T x, T y) method. In this example, I have renamed the parameters p1 and p2. Note that since I am comparing strings, I can simply call the String.CompareTo() method to actually perform the comparison.

Example 14.15 shows how the PersonComparer class is used to sort a list of Person objects.

14.15 ComparerSortDemo.cs

```
1     using System;
2     using System.Collections.Generic;
3
4     public class ComparerSortDemo {
5       public static void Main(){
6         List<Person> surrealists = new List<Person>();
7
8         Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
9         Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
10        Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
11        Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
12        Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
13        Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
14                               new DateTime(1887, 07, 28));
15
16        surrealists.Add(p1);
17        surrealists.Add(p2);
18        surrealists.Add(p3);
19        surrealists.Add(p4);
20        surrealists.Add(p5);
21        surrealists.Add(p6);
22
23        for(int i=0; i<surrealists.Count; i++){
24          Console.WriteLine(surrealists[i].FullNameAndAge);
25        }
26
27        surrealists.Sort(new PersonComparer());
28        Console.WriteLine("----------------------------------------");
29
30        for(int i=0; i<surrealists.Count; i++){
31          Console.WriteLine(surrealists[i].FullNameAndAge);
32        }
33
34      } // end Main()
35    } // end SortingListDemo
```

Referring to Example 14.15 — an overloaded version of the List<T>.Sort() method is called on line 27. This version of the Sort() method takes a Comparer<T> object as an argument. Figure 14-23 shows the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Sorting_with_Comparer>comparersortdemo
Rick  Miller 46
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
----------------------------------------
Andre  Breton 111
Henri-Robert-Marcel  Duchamp 120
Max  Ernst 116
Lee  Miller 100
Rick  Miller 46
Roland  Penrose 107

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Sorting_with_Comparer>
```

Figure 14-23: Results of Running Example 14.15

 © 2008 Rick Miller — All Rights Reserved

Converting A Collection Into An Array

It's often handy to convert the contents of a collection into an array. This holds especially true for collections that cannot be manipulated like lists. (*i.e.*, via indexers) Collection classes that implement the ICollection<T> interface provide a CopyTo() method that copies the contents of the collection into an existing array whose length is equal to that of the number of collection elements. Example 14.16 offers a short program that shows how to extract an array of KeyValuePair<TKey, TValue> values from a SortedDictionary<TKey, TValue> collection.

14.16 ConvertToArrayDemo.cs

```
1    using System;
2    using System.Collections.Generic;
3
4    public class ConvertToArrayDemo {
5      public static void Main(){
6        String[] names = { "Rick",
7                           "Sally",
8                           "Joe",
9                           "Bob",
10                          "Steve" };
11
12       SortedDictionary<String, String> quotes = new SortedDictionary<String, String>();
13
14       quotes.Add(names[0], "How Do You Do?");
15       quotes.Add(names[1], "When are we going home?");
16       quotes.Add(names[2], "I said go faster, man!");
17       quotes.Add(names[3], "Turn now! Nowww!");
18       quotes.Add(names[4], "The rain in Spain falls mainly on the plain.");
19
20       KeyValuePair<String, String>[] quote_array = new KeyValuePair<String, String>[quotes.Count];
21       quotes.CopyTo(quote_array, 0);
22
23       for(int i = 0; i<quote_array.Length; i++){
24         Console.WriteLine(quote_array[i].Key + " said: \"" + quote_array[i].Value + "\"" );
25       }
26
27     }// end Main()
28   } // end ConvertToArrayDemo class
```

Referring to Example 14.16 — first, the program creates an array of strings that's used to store five names. The SortedDictionary object is created on line 12. On lines 14 through 18, the names array provides the keys for each quote. Note that as each key/value pair is inserted into the SortedDictionary, it is inserted into its proper sorted order according to the key's value. (Ergo, classes used as keys must implement IComparable<T>) On lines 20 and 21, the array of sorted KeyValuePair elements is extracted into an array of KeyValuePairs equal in length to the collection's count. The quotes array is then used in the `for` statement beginning on line 23 to print the values of both the key and its associated value. In this case they are both strings. Figure 14-24 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Converting_To_Arrays>converttoarraydemo
Bob said: "Turn now! Nowww!"
Joe said: "I said go faster, man!"
Rick said: "How Do You Do?"
Sally said: "When are we going home?"
Steve said: "The rain in Spain falls mainly on the plain."

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Converting_To_Arrays>_
```

Figure 14-24: Results of Running Example 14.16

Quick Review

To make an object sortable, its class or structure type must implement the IComparable<T> interface. If you don't own the source code for a particular class or structure and wish to sort them using List<T>'s Sort() method, then you can implement a Comparer<T> object that defines how such objects are to be compared with each other.

It's often handy to convert the contents of a collection into an array. Collection classes that implement the ICollection<T> interface provide a CopyTo() method that is used to copy the contents of the collection into an existing array whose length is equal to that of the number of collection elements.

© 2008 Rick Miller — All Rights Reserved

Summary

The .NET collections framework can potentially save you a lot of time and hassle. It contains classes, structures, and interfaces designed to make it easy to manipulate collections of objects. The .NET 2.0 framework introduced generic collections and improved performance.

An *array* is a contiguous allocation of objects in memory. An array-based collection offers quick element access but slow element insertion, especially if the collection's underlying array must be resized and its contents shifted to accommodate the insertion.

A *linked list* consists of individual *nodes* linked to each other via references. To traverse a linked list, you must start at the beginning, or the end (head or tail) and follow each element to the next. Linked list-based collections can conserve memory space because memory need only be allocated on each object insertion. Insertions into linked list-based collections are relatively quick, but element access is relatively slow due to the need to traverse the list.

A *chained hash table* is a cross between an array and a linked list and allows element insertion with key/value pairs. A hash function performed on the key determines the value's location in the hash table. A collision is said to occur when two keys produce the same hash code. When this happens, the values are chained together in a linked list-like structure. A hash function that produces a uniform distribution over all the keys is a critical feature of a hash table.

A *red-black tree* is a self-balancing binary tree. Insertions into a red-black tree take some time because of element ordering and balancing operations. Element access time for a red-black tree-based collection is fairly quick.

"Old-school" style .NET collections classes store only object references and require casting when elements are retrieved. You should favor the use of generic collections as they offer strong element typing on insertion and retrieval as well as improved performance. The classes found in the System.Collections.ObjectModel namespace serve as the basis for user-defined custom collections. The System.Collections.Specialized namespace contains classes and structures designed to manage unique collections.

The List<T> generic collection class is the direct replacement for the ArrayList collection class. It's easy to use the List<T> generic collection. The T that appears between the left and right angle brackets represents the replacement type. You can think of the T as acting like a placeholder for a field in a mail-merge document. Anywhere the T appears, the type that's actually substituted for T will appear in its place when the collection object is created.

You can find the KeyedCollection<TKey, TItem> in the System.Collections.ObjectModel namespace. It is an abstract class and is meant to be used as the basis for a custom collection. It contains one abstract method, GetKeyForItem(), that you must override to properly extract the key from the item being inserted into the collection.

To make an object sortable, its class or structure type must implement the IComparable<T> interface. If you don't own the source code for a particular class or structure and wish to sort them using List<T>'s Sort() method, then you can implement a Comparer<T> object that defines how such objects are to be compared with each other.

It's often handy to convert the contents of a collection into an array. Collection classes that implement the ICollection<T> interface provide a CopyTo() method that is used to copy the contents of the collection into an existing array whose length is equal to that of the number of collection elements.

Skill-Building Exercises

1. **API Drill:** Access the MSDN online documentation and explore all four .NET collections namespaces. List each class, structure, and interface you find there and write a brief description of its purpose and use.

2. **API Drill:** Research the ICollection<T> interface and answer the following question: What exception will be thrown if the number of collection elements exceeds the size of the array supplied via the CopyTo() method?

3. **API Drill:** What methods are available in the List<T> class that let you extract its elements into an array?

4. **Programming Drill:** Compile and run all the example programs presented in this chapter and note the results. Modify the programs as you see fit to utilize more of the methods provided by each collection class.

 © 2008 Rick Miller — All Rights Reserved

5. **Programming Drill:** Explore the System.Collections.Generic and System.Collections.Specialized namespaces, select several collections not discussed in this chapter and use each in a short program.

Suggested Projects

1. **Employee Management Application:** Create a GUI-based application that lets you create and manage Employees. Use the Person, IPayable, Employee, HourlyEmployee, and SalariedEmployee code originally presented in Chapter 11. Modify the Employee class to implement the IComparable<T> interface or, alternatively, create a Comparer<T> class that compares two Employee objects. Your program should allow you to create HourlyEmployee and SalariedEmployee objects and insert them into a generic collection that holds Employee type objects. Each time you create a new employee display a sorted list of employees.

2. **Collectables Application:** Are you a collector? Do you have lots of books, CDs, DVDs, or antique typewriters? Do you have a tiny little spoon from every state of the Union, or a book of matches from every restaurant or nightclub you've visited? Whatever your passion, write a program that helps you manage your collection. Create a class that captures the properties of your particular collectable. Create a GUI-based application that lets you enter information about each item in your collection. **Extra credit:** Add the capability to upload an image of the item and persist the collection object to disk so you don't lose the data you entered. **Note:** The serialization of objects to disk is discussed in Chapter 17 - File I/O.

Self-Test Questions

1. What must you do to elements retrieved from an "old-school" collection before accessing its members or performing an operation on it other than those defined by System.Object?

2. What are the general performance characteristics of an array-based collection?

3. What are the general performance characteristics of a linked list-based collection?

4. What are the general performance characteristics of a hash table-based collection?

5. What are the general performance characteristics of a red-black tree-based collection?

6. What does the T represent in a generic collection?

7. What interface must a class or structure implement before it can be sorted by the List<T>.Sort() method?

8. How can you sort objects using the List<T>.Sort() method if you don't have access to a class's source code?

9. Why would you want to use generic collections vs. earlier "old-school" collections?

10. What method do you use to extract a collection's elements into an array?

© 2008 Rick Miller — All Rights Reserved

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

John Franco. *Red-Black Tree Demonstration Applet*. [http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html]

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

Rick Miller, Raffi Kasparian. *Java For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-05-2

Notes

© 2008 Rick Miller — All Rights Reserved

Chapter 15

Contax T / Kodak Tri-X

Happy Mom

Exceptions: Writing Fault-Tolerant Software

Learning Objectives

- *Define the term "exception"*
- *Trace the Exception class hierarchy*
- *List and describe the runtime exceptions*
- *List and describe the properties of the Exception class*
- *Explain the difference between an application exception vs. a runtime exception*
- *Describe the exception handling mechanism supported by the .NET Common Language Runtime (CLR)*
- *Create your own custom exception classes*
- *Translate low-level exceptions into high-level exceptions*
- *State the purpose of a try/catch block*
- *State the purpose of a finally block*
- *Describe an appropriate use of a try/finally block*
- *State the proper order in which exceptions should be caught*
- *Determine what exceptions a method throws*
- *Create and throw an exception*

© 2008 Rick Miller — All Rights Reserved

Introduction

C# and the .NET Common Language Runtime use exceptions to indicate that an error 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, C# programs are under a constant threat of running into some kind of trouble. As program complexity grows, so does the possibility that something will go wrong during its execution.

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 .NET API) and how to create your own custom exceptions. You will be formally introduced to the Exception class, try/catch/finally blocks, and the `throw` keyword. You will also learn the difference between *application* and *runtime* 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!

What Is An Exception

An exception is an error condition or abnormal event that occurs during program execution. By error condition, I mean a fault in the technical execution of a particular statement within your program, not bad programming logic or poor algorithms. (Although these may ultimately result in an exception!) Some examples of error conditions include the following:

- Trying to access an element beyond the bounds of an array.
- Running out of memory during program execution.
- Network communications problems.
- Unverifiable executable code modules.

Exceptions can occur in the code you write or in external code that your program calls during execution. Such code might be located in a dynamic link library (dll).

.NET CLR Exception Handling Mechanism

When an exception occurs in a program, the .NET runtime handles the problem by creating an exception object, populating it with information related to the error condition that occurred, and then passing it on to any fault-handling code that has been designated to respond to that particular type of exception. This process is referred to as *throwing* an exception.

Any code that may result in an exception condition is placed in a *protected block*. (*i.e.,* a try block) That is, if you want to provide fault handler code, (*i.e.,* a `catch` block) then you must place the suspect code in a `try` block and then handle or *catch* the thrown exception in a `catch` block.

Unhandled Exceptions

C# does not force you to use `try/catch` blocks. Unlike Java, there are no checked exceptions. The .NET runtime will try its best to find a fault handler for an exception thrown in an unprotected block of code. It does this by passing the exception object up the method call stack to see if the calling method provided a fault handler. If none is

© 2008 Rick Miller — All Rights Reserved

found, the .NET runtime generates an UnhandledException event. If there are no event handlers for this Unhandled-Exception event, the .NET runtime writes a dump of the stack trace to the console and the program exits

The Exception Information Table

The .NET runtime keeps track of an executable's exception information in a data structure referred to as an *exception information table* as Figure 15-1 illustrates.

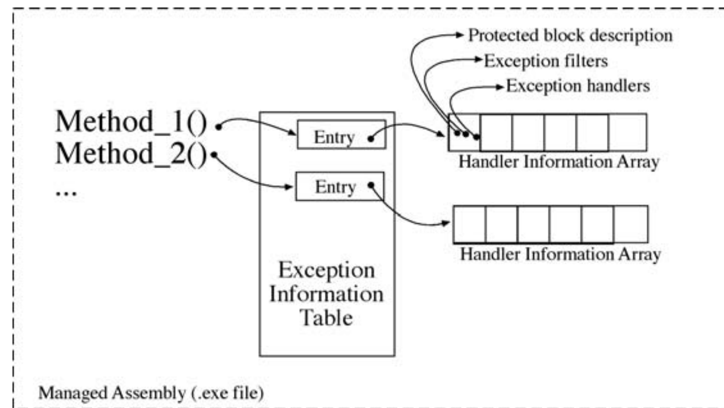


Figure 15-1: Exception Information Table

Referring to Figure 15-1 — all methods in an executable (*i.e.*, a .exe managed assembly) have an entry in this table that contains an array of exception-handling information. This array will be empty if the method does not have any `try/catch/finally` blocks. Each entry of this array contains the following information:

- A protected code block description.
- Associated exception filters.
- Exception handlers (*i.e.*, `catch` blocks, `finally` blocks, and type-filter handlers)

When an exception occurs, the .NET runtime searches the array for the first protected block of code that contains the currently executing instruction and that also has an associated exception handler. If it finds a match, the .NET runtime creates the appropriate exception-type object and then executes any `finally` or fault statements before passing the exception on to the appropriate `catch` block.

Quick Review

An *exception* is an error condition or abnormal event that occurs during program execution.

The .NET runtime handles exceptions by creating an exception object, populating it with information related to the error condition that occurred, and then passing it on to any fault-handling code that has been designated to respond to that particular type of exception. This process is referred to as *throwing* an exception.

Protected code is located in a `try` block. Error handlers are placed in a `catch` block.

Each executable file loaded into the .NET runtime contains an *exception information table*. All methods contained within the executable file have an entry in the exception information table. When an exception occurs, a method's corresponding handler information array is searched for any associated error handlers. If none are found, the exception is propagated up the method call stack to search the calling method's handler information array. If no handler methods are found, the .NET runtime dumps the stack trace to the console and the program terminates.

Exception Class Hierarchy

The .NET Framework provides a hierarchy of exception classes that derive from the System.Exception class as Figure 15-2 illustrates. Referring to Figure 15-2 — two primary types of exceptions derive from the Exception class: SystemException and ApplicationException. You'll find these exception types in the System namespace.

© 2008 Rick Miller — All Rights Reserved

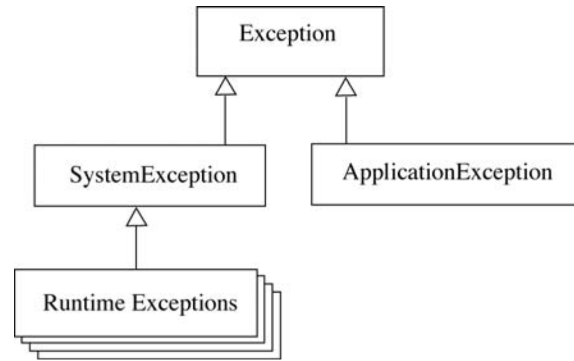


Figure 15-2: Exception Class Hierarchy

Application vs. Runtime Exceptions

Runtime exceptions derive from the SystemException class. Runtime exceptions can occur for two primary reasons: 1) failed runtime checks such as trying to access an element past the bounds of an array (IndexOutOfRangeException) or trying to access members via a reference that points to a null value (NullReferenceException), and 2) severe error conditions that occur within the runtime execution environment such as running out of memory (OutOfMemoryException) or when the execution engine has been corrupted or encounters missing data (EngineExecutionException) to name two examples. You can easily recover gracefully from an exception thrown due to a failed runtime check by catching and handling the exception, but there's not much you can do to recover from a fatal runtime exception. This is mainly because you don't know when or where in your code such an exception may occur.

An *application exception* is any exception thrown by a running program that's not a runtime exception. Application exceptions derive from either the ApplicationException class or directly from the Exception class itself. In fact, Microsoft admits that deriving custom exceptions from the ApplicationClass doesn't provide any benefit over deriving directly from Exception, so they suggest deriving all custom exceptions directly from the Exception class.

Runtime Exception Listing

Table 15-1 lists a few of the standard runtime exceptions you'll need to consider and handle.

Runtime Exception	Description
IndexOutOfRangeException	This exception is thrown if you try to access an array element that lies outside the bounds of the array. For example, assume you have an array of integers with Length 5. The array has 5 elements, 0 through Length-1. Any attempt to access an array element less than 0 or greater than 4 causes an IndexOutOfRangeException.
NullReferenceException	This exception is thrown if you try to access an object's members via a null reference. A null reference is a reference that points to nothing.
AccessViolationException	This exception is thrown if you try to access memory via an invalid pointer in unmanaged code.
InvalidOperationException	This exception is thrown if the object upon which you call a method has entered an invalid state. This usually occurs if you modify a collection and then call the Enumerator.GetNext() method on that collection. You can easily and unknowingly make this mistake by attempting to modify a collection in the body of a `foreach` loop where enumerator semantics are hidden from you.

Table 15-1: Runtime Exceptions — Partial Listing

© 2008 Rick Miller — All Rights Reserved

Runtime Exception	Description
ArgumentNullException	This exception is thrown if a method does not allow one or more of its parameters to be null when the method is called.
ArgumentOutOfRangeException	This exception is thrown if one or more of a method's parameter's value falls outside of its valid value range.

Table 15-1: Runtime Exceptions — Partial Listing

Determining What Exceptions A .NET Framework Method Throws

To determine what exceptions a .NET Framework method throws, you need to consult the documentation.

Exceptions the method may throw are listed in the **Exceptions** section.

Figure 15-3: Getting Exception Information from MSDN

Referring to Figure 15-3 — you'll find a list of the exceptions a method can throw in the **Exceptions** section of the method's description page. In most cases, methods can potentially throw many different types of exceptions. It's always a good idea to check the Exception section to ensure you're properly handling any exceptions that may be thrown during a method's execution.

Quick Review

Two primary types of exceptions derive from the Exception class: *SystemException* and *ApplicationException*. Runtime exceptions can occur for two primary reasons: 1) failed runtime checks, and 2) severe error conditions that occur within the runtime execution environment. An application exception is any exception thrown by a running program that's not a runtime exception. Application exceptions derive from either the ApplicationException class or directly from the Exception class itself. Consult the **Exceptions** section of a method's MSDN documentation page to learn what exceptions the method might throw.

© 2008 Rick Miller — All Rights Reserved

Exception Class Properties

A thrown exception conveys a lot of helpful information. In this section, I want to describe the properties of the Exception class. Understanding what information an exception object contains helps you to better understand what went wrong in your program. Table 15-2 lists the Exception class's public properties.

Property	Read/Write	Type	Description
Data	Readonly	Dictionary	The Data property is **readonly** and gets a collection that implements the IDictionary interface. The Data collection is by default empty. You are free to store additional information about an exception in the Data collection in the form of key/value pairs.
HelpLink	Read/Write	String	The HelpLink property is **read/write** and is used to get or set a string that represents a link to a help file or other resource that provides information about the exception.
InnerException	Readonly	Exception	The InnerException property is **readonly** and gets the exception that caused the current exception. The InnerException property is set via a constructor at the time the exception object is created.
Message	Readonly	String	Message is a **readonly** property that gets a string containing information about the current exception. The Message property is set via a constructor.
Source	Read/Write	String	The Source property is **read/write** and is used to get and set the name of the application or object that caused the exception.
StackTrace	Readonly	String	The StackTrace property is **readonly** and is used to get a string representation of the call stack at the moment the exception was generated. The StackTrace property is set automatically by the .NET runtime.
TargetSite	Readonly	MethodBase	The TargetSite property is **readonly** and is used to get information about the method that threw the exception. The TargetSite property returns a MethodBase object. The MethodBase class contains properties that describe all aspects of a particular method

Table 15-2: Exception Class Public Properties

Referring to Table 15-2 — in most cases, a property is readonly and is used to get information about a particular exception. Initializing a readonly exception property is usually done via one of the Exception class's overloaded constructors when the exception object is created. The Data property returns a dictionary collection which will always initially be empty. Populating the Data collection with relevant information is left to your discretion. The InnerException property is used when translating from one exception type to another or from low-level exceptions to higher-level exception abstractions that you create for your particular application. I cover this topic in more detail in the Creating Custom Exceptions section. The TargetSite property returns a MethodBase object, which conveys a lot of information about the method that threw the exception. You will see all of these properties in action in the next two sections.

Quick Review

The Exception class contains seven properties that are used to get information about why an exception was thrown. Most of the properties are readonly and can only be set via a constructor call when an exception object is created.

© 2008 Rick Miller — All Rights Reserved

Creating Exception Handlers: Using Try/Catch/Finally Blocks

To catch and handle exceptions, you need to place exception-throwing code in a `try` block and catch the resulting exception, if and when it gets thrown, in one or more `catch` blocks where error-handling code is placed to properly deal and recover from the exception. Optionally, you can add a `finally` block whose code will always be executed regardless of whether or not an exception is thrown. This section explains the use of `try/catch/finally` blocks in detail, offers a few pointers on defensive coding, and explains in what order you must catch exceptions when using multiple `catch` blocks.

Using A Try/Catch Block

The use of a `try/catch` block is straightforward. You place any code that may throw an exception within the body of the `try` block and follow it with one or more `catch` blocks. The code that resides within the body of a `try` block is referred to as *protected code*.

A `catch` block contains error-handling code that is executed if an exception of the type the `catch` block is waiting for is thrown. Example 15.1 demonstrates the use of a `try/catch` block to protect against possible problems associated with processing console arguments.

15.1 ConsoleArgs.cs

```
1    using System;
2
3    public class ConsoleArgs {
4       public static void Main(String[] args){
5          try{
6
7             Console.WriteLine(args[0]);
8
9          }catch(IndexOutOfRangeException e){
10            Console.WriteLine("HelpLink:" + e.HelpLink);
11            Console.WriteLine("Message:" + e.Message);
12            Console.WriteLine("Source:" + e.Source);
13            Console.WriteLine("TargetSite:" + e.TargetSite.Name);
14            Console.WriteLine("StackTrace:" + e.StackTrace);
15         }
16      }
17   }
```

Referring to Example 15.1 — this program prints to the console the first argument of the args array. The problem with this program is that it may be run without any arguments and therefore the args array will be empty. Any attempt to access an element of an args array that contains no elements results in an IndexOutOfRangeException. To handle this possibility, the code on line 7 is placed within the protection of a `try` block.

The `catch` block that begins on line 9 specifically targets the IndexOutOfRangeException. A parameter of this type named *e* is declared for use within the body of the `catch` block. The code within the body of the `catch` block demonstrates the use of the various Exception properties. In practice, you don't always need to print this much information about an exception to the console.

Figure 15-4 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\ConsoleArgs>consoleargs Hello!
Hello!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\ConsoleArgs>consoleargs
HelpLink:
Message:Index was outside the bounds of the array.
Source:ConsoleArgs
TargetSite:Main
StackTrace:   at ConsoleArgs.Main(String[] args)

C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\ConsoleArgs>_
```

Figure 15-4: Results of Running Example 15.1

© 2008 Rick Miller — All Rights Reserved

First Line of Defense: Use Defensive Coding

It's often a good idea to avoid the possibility of throwing an exception in the first place. It's good coding practice to apply defensive coding techniques to detect the possibility of an exception condition and handle the situation accordingly. Note that this is not always possible, especially for certain types of runtime exceptions.

Example 15.2 shows how a slight modification to the ConsoleArgs code can avoid the possibility of throwing an IndexOutOfRangeException altogether.

15.2 ConsoleArgs.cs (Mod 1)

```
1    using System;
2
3    public class ConsoleArgs {
4       public static void Main(String[] args){
5          try{
6
7             if(args.Length > 0){
8                Console.WriteLine(args[0]);
9             }
10
11         }catch(IndexOutOfRangeException e){
12            Console.WriteLine("HelpLink:" + e.HelpLink);
13            Console.WriteLine("Message:" + e.Message);
14            Console.WriteLine("Source:" + e.Source);
15            Console.WriteLine("TargetSite:" + e.TargetSite.Name);
16            Console.WriteLine("StackTrace:" + e.StackTrace);
17         }
18      }
19   }
```

Referring to Example 15.2 — the `if` statement beginning on line 7 checks to see if args.Length is greater than zero. If it is, it must contain at least one argument string, otherwise, it's not accessed.

Figure 15-5 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\ConsoleArgs_Mod_1>consoleargs hello
hello

C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\ConsoleArgs_Mod_1>consoleargs

C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\ConsoleArgs_Mod_1>_
```

Figure 15-5: Results of Running Example 15.2

Using Multiple Catch Blocks

If your code can potentially throw several types of exceptions you can add multiple `catch` blocks, one for each exception type. The rule to follow when using multiple `catch` blocks is to catch the most specific exception(s) first, and catch the most general exception(s) last. Example 15.3 offers an example of using multiple `catch` blocks. This program converts command-line arguments to integers, adds them, and prints the sum to the console.

15.3 CommandLineAdder.cs

```
1    using System;
2
3    public class CommandLineAdder {
4       public static void Main(String[] args){
5          try{
6             int total = 0;
7             for(int i=0; i<args.Length; i++){
8                total += Int32.Parse(args[i]);
9             }
10            Console.WriteLine("You entered {0} arguments and their total comes to {1}", args.Length, total);
11         }catch(FormatException){
12            Console.WriteLine("One or more arguments failed to convert to an integer!");
13         }catch(IndexOutOfRangeException){
14            Console.WriteLine("No command line arguments entered!");
15         }catch(Exception){
16            Console.WriteLine("The program encountered an unknown problem...");
17         }
18      }
19   }
```


© 2008 Rick Miller — All Rights Reserved

Referring to Example 15.3 — the `try` block in this example has three accompanying `catch` blocks which handle a range of possible exceptions including the granddaddy of them all, Exception. Note that you're not required to declare a formal parameter in the `catch` block if you're not planning on manipulating the exception object. Figure 15.6 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\CommandLineAdder>commandlineadder 45 23 33
You entered 3 arguments and their total comes to 101

C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\CommandLineAdder>commandlineadder 45 23 33 bad
One or more arguments failed to convert to an integer!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\CommandLineAdder>
```

Figure 15-6: Results of Running Example 15.3

Using A Finally Block

Referring to Example 15.3 and Figure 15-6 above, note that when a bad string (*i.e.,* one that does not parse to an integer) is entered on the command line, an exception is thrown and line 10 is skipped. All code that comes after an exception-throwing statement in a `try` block is skipped. Depending on the nature of the application, this may or may not be acceptable.

In certain programming situations, you will want to ensure that a certain piece of code executes in all cases regardless of whether or not an exception is thrown. A `finally` block is used for just this purpose. Example 15.4 demonstrates the use of a finally block.

15.4 CommandLineAdder.cs (Mod 1)

```
1    using System;
2
3    public class CommandLineAdder {
4      public static void Main(String[] args){
5        try{
6          int total = 0;
7          for(int i=0; i<args.Length; i++){
8            total += Int32.Parse(args[i]);
9          }
10         Console.WriteLine("You entered {0} arguments and their total comes to {1}", args.Length, total);
11       }catch(FormatException){
12         Console.WriteLine("One or more arguments failed to convert to an integer!");
13       }catch(IndexOutOfRangeException){
14         Console.WriteLine("No command line arguments entered!");
15       }catch(Exception){
16         Console.WriteLine("The program encountered an unknown problem...");
17       }finally{
18         Console.WriteLine("Thank you for using Command Line Adder!");
19       }
20     }
21   }
```

Referring to Example 15.4 — the `finally` block starts on line 17 and simply ensures that no matter what happens when the program runs, the "Thank you..." message gets printed to the console. You will see plenty of good uses for a `finally` block when you study file input/output (I/O) and network programming. Figure 15-7 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\FinallyBlock>commandlineadder 1 2 3
You entered 3 arguments and their total comes to 6
Thank you for using Command Line Adder!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\FinallyBlock>commandlineadder a b c
One or more arguments failed to convert to an integer!
Thank you for using Command Line Adder!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\FinallyBlock>
```

Figure 15-7: Results of Running Example 15.4

© 2008 Rick Miller — All Rights Reserved

Quick Review

To catch and handle exceptions, you need to place exception-throwing code in a `try` block and catch the resulting exception, if and when it gets thrown, in one or more `catch` blocks where error-handling code is placed to properly deal and recover from the exception. Optionally, you can add a `finally` block whose code will always be executed regardless of if an exception is thrown.

All code within a `try` block that follows an exception-throwing statement is skipped. Place critical resource releasing code in a `finally` block.

If code can potentially throw several types of exception, add multiple `catch` blocks, one for each exception type. The rule to follow when using multiple `catch` blocks is to catch the most specific exception(s) first, and catch the most general exception(s) last.

Creating Custom Exceptions

Although the .NET Framework API offers lots of different types of exception classes, you'll eventually have a need to create your own custom exceptions to better model the types of things that can go wrong in your application. This section shows you how to create custom exception classes by extending the Exception class. I also show you how to manually throw an exception and how to translate low-level exceptions into higher-level exceptions.

Extending The Exception Class

As you learned previously in this chapter, although there exists an ApplicationException class, Microsoft suggests that you instead derive custom exception classes directly from the Exception class. At a minimum, you'll want to end the name of your custom exception class with the word "Exception" and provide an implementation for each of the three common Exception constructors. Example 15.5 gives the code for a custom exception named SurrealistNotFoundException, which is used later in this section to demonstrate how to manually throw an exception.

15.5 SurrealistNotFoundException.cs

```
1    using System;
2
3    public class SurrealistNotFoundException : Exception {
4
5       public SurrealistNotFoundException() : base("Surrealist not found!") { }
6
7       public SurrealistNotFoundException(String message) : base(message) { }
8
9       public SurrealistNotFoundException(String message, Exception inner_exception)
10                                       : base(message, inner_exception) { }
11
12   }
```

Referring to Example 15.5 — this custom exception provides implementations for the three basic Exception constructors. The default constructor that begins on line 3 supplies a default message. The rest of the code is fairly straightforward and easy to follow. Let's now see how this custom exception might be used in a program. Examples 15.6 and 15.7 give the code for a short program that allows users to look up the names of famous surrealists.

15.6 SurrealistBank.cs

```
1    /***************************************************
2      This program depends on the Person class.
3    ***************************************************/
4
5    using System;
6    using System.Collections.Generic;
7
8    public class SurrealistBank {
9       Dictionary<String, Person> surrealists;
10
11      public SurrealistBank() {
12         surrealists = new Dictionary<String, Person>();
13         InitializeDictionary();
14   }
15
16      private void InitializeDictionary() {
17         Person p1 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
18         Person p2 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
```

 © 2008 Rick Miller — All Rights Reserved

```
19        Person p3 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
20        Person p4 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
21        Person p5 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
22                               new DateTime(1887, 07, 28));
23        surrealists.Add(p1.LastName, p1);
24        surrealists.Add(p2.LastName, p2);
25        surrealists.Add(p3.LastName, p3);
26        surrealists.Add(p4.LastName, p4);
27        surrealists.Add(p5.LastName, p5);
28      }
29
30      public Person LookUp(String last_name) {
31        Person p = null;
32        try {
33          if (surrealists.TryGetValue(last_name, out p)) {
34            return p;
35            }
36          else {
37              throw new SurrealistNotFoundException("That name is not in the surrealist collection!");
38          }
39        }
40        catch (ArgumentNullException ane) {
41          throw new SurrealistNotFoundException("A null string name was entered!", ane);
42        }
43      }
44    } // end SurrealistBank class definition
```

Referring to Example 15.6 — as stated in the comments at the top of the source file, this class depends on the Person class. When you compile this example, be sure to copy over the Person class from one of the examples given in Chapter 14.

The SurrealistBank class creates a Dictionary and populates it with Person objects. Its LookUp() method searches the dictionary for a matching key value via the Dictionary.TryGetValue() method. Note how the Person reference p is passed as an `out` parameter to the method call. The TryGetValue() method returns the boolean value *true* if there's a key match; *false* otherwise. If there's a match, the LookUp() method returns p.

15.7 MainApp.cs

```
1     using System;
2
3     public class MainApp {
4       public static void Main() {
5         SurrealistBank sb = new SurrealistBank();
6         String input = "Go";
7         while (!input.Equals("Quit")) {
8           Console.Write("Please enter a name to lookup or 'Quit' to exit the program: ");
9           input = Console.ReadLine();
10            try {
11              Console.WriteLine();
12              Console.WriteLine(sb.LookUp(input));
13            } catch (SurrealistNotFoundException snfe) {
14                Console.WriteLine(snfe.Message);
15              }
16        }//end while
17      }
18    }
```

Referring to Example 15.7 — when the program executes, the input string is initialized to "Go". The `while` loop beginning on like 7 repeats until the user enters "Quit". All strings read from the console are presented to the sb.LookUp() method, which may throw a SurrealistNotFoundException. Figure 15-8 shows the results of running this program.

Manually Throwing An Exception With The throw Keyword

Referring to Example 15.6 — note again that if there's a match, the LookUp() method returns p, otherwise, it throws the SurrealistNotFoundException. It does this by using the `throw` keyword in conjunction with the creation of a new SurrealistNotFoundException object.

Translating Low-Level Exceptions Into High-Level Exceptions

Referring again to Example 15.6 — the second `catch` block beginning on line 40 handles the ArgumentNullException. This exception may be thrown by the Dictionary.TryGetValue() method if the supplied key object is null. In this example, the ArgumentNullException supplied by the .NET Framework is translated into a higher-level excep-

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\CustomException>mainapp
Please enter a name to lookup or 'Quit' to exit the program: Miller

Lee  Miller is a FEMALE who is 100 years old.
Please enter a name to lookup or 'Quit' to exit the program: Duchamp

Henri-Robert-Marcel  Duchamp is a MALE who is 120 years old.
Please enter a name to lookup or 'Quit' to exit the program: Penrose

Roland  Penrose is a MALE who is 107 years old.
Please enter a name to lookup or 'Quit' to exit the program: Picasso

That name is not in the surrealist collection!
Please enter a name to lookup or 'Quit' to exit the program: Quit

That name is not in the surrealist collection!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter15\CustomException>
```

Figure 15-8: Results of Running Example 15.7

tion abstraction by throwing a SurrealistNotFoundException and using the ArgumentNullException object to set its InnerException property via the constructor call.

Quick Review

Create a *custom exception* by extending the Exception class and providing implementations for each of its three common constructors. End the name of your custom exception with the word "Exception". Use the `throw` keyword to throw an exception object.

Documenting Exceptions

You must consult the MSDN, or your locally generated documentation, to learn what exceptions, if any, a method might throw upon execution. To document an exception use the <exception> tag. Example 15.8 shows how the <exception> tag is used to document the exception thrown by the SurrealistBank.LookUp() method.

15.8 Documented SurrealistBank.LookUp() Method

```
1     /// <summary>
2     /// Searches the SurrealistBank's dictionary for the given name. Returns a populated
3     /// Person object if there's a match. Throws a SurrealistNotFoundException if no
4     /// match is found or if a null string is used as an argument.
5     /// </summary>
6     /// <param name="last_name"> A string representing a surrealist's last name.</param>
7     /// <returns>Fully populated Person object.</returns>
8     /// <exception cref="SurrealistNotFoundException"></exception>
9     public Person LookUp(String last_name) {
10       Person p = null;
11       try {
12          if (surrealists.TryGetValue(last_name, out p)) {
13             return p;
14          }
15          else {
16             throw new SurrealistNotFoundException("That name is not in the surrealist collection!");
17          }
18       } catch (ArgumentNullException ane) {
19             throw new SurrealistNotFoundException("A null string name was entered!", ane);
20          }
21    }
```

Referring to Example 15.8 — on line 8, the <exception> tag is used to document the possibility that the LookUp() method may throw the SurrealistNotFoundException. The `cref` attribute provides a link to the Surrealist-NotFoundException class. The name of the class should be fully namespace qualified.

© 2008 Rick Miller — All Rights Reserved

Summary

An *exception* is an error condition or abnormal event that occurs during program execution.

The .NET runtime handles exceptions by creating an exception object, populating it with information related to the error condition that occurred, and then passing it on to any fault-handling code that has been designated to respond to that particular type of exception. This process is referred to as "throwing" an exception.

Protected code is located in a `try` block. Error handlers are placed in a `catch` block.

Each executable file loaded into the .NET runtime contains an *exception information table*. All methods contained within the executable file have an entry in the exception information table. When an exception occurs, a method's corresponding handler information array is searched for any associated error handlers. If none are found, the exception is propagated up the method call stack to search the calling method's handler information array. If no handler methods are found, the .NET runtime dumps the stack trace to the console and the program terminates.

Two primary types of exceptions derive from the Exception class: *SystemException* and *ApplicationException*. *Runtime exceptions* occur for two primary reasons: 1) failed runtime checks, and 2) severe error conditions that occur within the runtime execution environment. An *application exception* is any exception thrown by a running program that's not a runtime exception. Application exceptions derive from either the ApplicationException class or directly from the Exception class itself. Consult the Exceptions section of a method's MSDN documentation page to learn what exceptions a method might throw.

To catch and handle exceptions, you need to place exception-throwing code in a `try` block and catch the resulting exception in one or more `catch` blocks where error-handling code is placed to properly deal and recover from the exception. Optionally, you can add a `finally` block whose code is always executed regardless of if an exception is thrown or not.

All code within a `try` block that follows an exception-throwing statement is skipped. Place critical resource releasing code in a `finally` block.

If your code can throw several types of exceptions, you can add multiple `catch` blocks, one for each exception type. The rule to follow when using multiple `catch` blocks is to catch the most specific exception(s) first, and catch the most general exception(s) last.

Create a *custom exception* by extending the Exception class and providing implementations for each of its three common constructors. End the name of your custom exception with the word "Exception". Use the `throw` keyword to throw an exception object.

Skill-Building Exercises

1. **API Drill:** Access the MSDN online documentation and explore the exception classes defined within the System namespace. Note their purpose and under what circumstances they might be thrown.

2. **API Drill:** Access the MSDN online documentation and explore the exception classes defined within the System.Collection and System.Collection.Generic namespaces. Note their purpose and under what circumstances they might be thrown.

3. **API Drill:** Access the MSDN online documentation and explore the exception classes defined within the System.IO namespace. Note their purpose and under what circumstances they might be thrown.

4. **API Drill:** Access the MSDN online documentation and explore the exception classes defined within the System.Net namespace. Note their purpose and under what circumstances they might be thrown.

© 2008 Rick Miller — All Rights Reserved

Suggested Projects

1. None

Self-Test Questions

1. What are the two meanings of the term *exception*?

2. How does the .NET runtime deal with exceptions?

3. How does the .NET runtime keep track of protected code and associated exception handlers?

4. What are the two primary types of exceptions defined my the .NET Framework?

5. What's the difference between a RuntimeException and an ApplicationException?

6. In what cases might a RuntimeException be thrown?

7. (T/F) Microsoft recommends extending the ApplicationException class to create a custom exception?

8. What happens to the code in a `try` block that follows a statement that just threw an exception?

9. Where should you place critical resource-freeing code or code that must be executed regardless of whether or not an exception is thrown?

10. What document tag do you use to indicate what exceptions a method can throw?

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

© 2008 Rick Miller — All Rights Reserved

Notes

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Chapter 16

Nikon F3HP / Zoom-Nikkor 35-105 / Kodak Tri-X

Operation – USS Norfolk SSN 714

Multithreaded Programming

Learning Objectives

- *Define the terms "process" and "thread"*
- *Explain the difference between a process and a thread*
- *Explain how time-slicing works*
- *State the definition of the term "multithreading"*
- *List and describe the steps and components required to create a multithreaded program*
- *Create and start threads in a program with the Thread class*
- *Pass arguments to managed threads using ParameterizedThreadStart delegates*
- *Explain the difference between foreground and background threads*
- *Use the BackgroundWorker class to create multithreaded programs*
- *Use delegates to make asynchronous method calls*
- *Pass arguments to and obtain results from asynchronous method calls*
- *Create background threads using the ThreadPool class*
- *Demonstrate your ability to create multithreaded programs*

© 2008 Rick Miller — All Rights Reserved

Introduction

It's time now to add another invaluable tool to your programming toolbelt, and that is the ability to write multithreaded programs. While the term *multithreaded programming* may sound complicated, it is in reality quite easy to do in C# .NET.

In this chapter, I will explain how multithreading works on your typical, general-purpose computer. You'll learn about the relationship between a process and its threads and how an operating system manages thread execution. I'll then show you how to use the Thread class in your programs to create and start managed threads. Next, I'll show you how you can simplify the creation and management of multiple threads with the help of the BackgroundWorker class. Also, I'll show you how to use ThreadPool threads and how to run any method asynchronously with the help of delegates.

As is the case with any tool, there's a right way to use it and a wrong way. Multithreading, applied thoughtlessly, will render your programs overly complicated, sluggish, unresponsive, and buggy. But, when used with care, multithreading can significantly increase your application's performance, giving it that hard-to-describe-but-you-know-it-when-you-see-it feeling of professionalism.

Before getting started I need to add a caveat. While I present a lot of material in this chapter, I make no attempt to cover all aspects of multithreaded programming. To do so would bore you to death, and in fact, as it turns out, a lot of what you can do with threads you shouldn't do. Instead, I will focus on those topics that give you a lot of bang for your buck to get you up and running as quickly as possible with multithreaded programming. In some cases, I have postponed the discussion of more obscure threading topics until later in the book where their presentation is more appropriate.

As usual, I recommend that if you want to dive deeper into threads and multithreaded programming consult one of the excellent references I've listed at the end of the chapter.

Multithreading Overview: The Tale Of Two Vacations

As I write this it's approaching the end of October in Northern Virginia. I always get a hankering to go on a vacation right about now. Let's take a look at the concept of vacation from a thread's point of view.

Single-Threaded Vacation

Imagine for a moment you're on vacation, trying to relax on the squeaky white sand of a sun-drenched tropical beach. Your job, since you are on vacation, is to relax, and as you start to drift off for a snooze you get thirsty. You are the only one on the beach and the bar is a mile away! You get up and walk to the bar and buy a drink, no, better make that two drinks, and walk back to your lounge chair. Now you start to relax again, until you get hungry. The grill is a mile in the other direction, so you get up again and walk to the grill. What you really want to do is relax and enjoy the beach, but what you ended up doing was a little relaxing, some drink fetching, and some food hunting. Eventually you'll get back to relaxing. After all, you're on single-threaded vacation.

Multithreaded Vacation

Now imagine that you're on multithreaded vacation. Again, your job is to relax on the beach. This time, however, when you get thirsty, you ask the wait staff to please bring you a drink, which they immediately set out to do, while you immediately return to relaxing. When you get hungry, you again summon the wait staff and off they go to fetch you a little somethin' somethin' from the grill. You immediately return to relaxing. Multithreaded vacation is so much better! If you've ever returned from a vacation and felt like you needed a vacation, you probably didn't take a multithreaded vacation because you were never allowed to relax!

 © 2008 Rick Miller — All Rights Reserved

The Relationship Between A Process And Its Threads

In the tale of two vacations above, you could think of yourself as being a process: the “Relax” process. On a computer, the operating system loads and starts services and applications. Each service or application runs as a separate *process* on the machine. (**Note:** A *service* is a special type of Windows application that runs solely in the background with limited or no user interaction.) Figure 16-1 shows a list of applications running on my machine as I write these words. Figure 16-2 shows a list of processes.

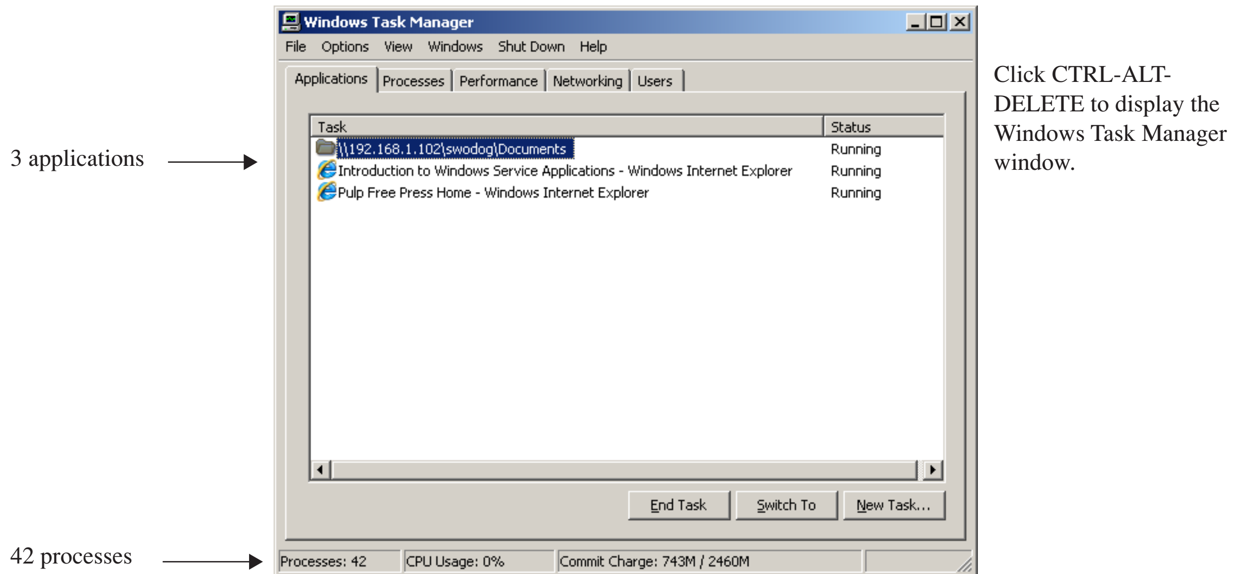


Figure 16-1: List of Running Applications

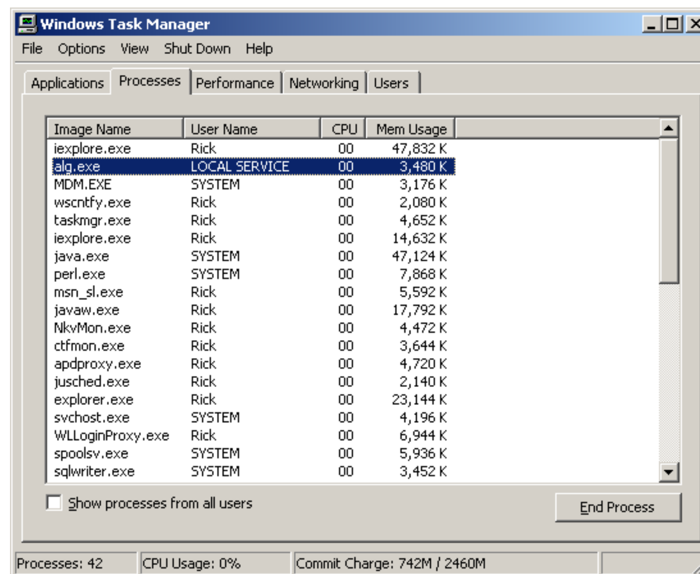


Figure 16-2: Partial List of Processes Running on the Same Computer

Referring to figures 16-1 and 16-2 — there are quite a few more processes actually running than there are applications. Many of the processes are background operating system processes that are started automatically when the computer powers up. There are two databases running: MS/SQL Server Express and Oracle 10G. You can also see in the process list that Java (java.exe) and a Perl interpreter (perl.exe) are running along with the Windows Desktop Explorer (explorer.exe) and two instances of Internet Explorer (iexplore.exe). Each one of these processes is isolated

© 2008 Rick Miller — All Rights Reserved

from the others meaning that each process has an allocated memory space all to itself. The management of this process memory space is left to the operating system.

A process consists of one or more *threads of execution*, referred to simply as *threads*. A process always consists of at least one thread, the *Main thread*, (the Main() method's thread of execution) which starts running when the process begins execution. A *single-threaded process* contains only one thread of execution. A *multithreaded process* contains more than one thread. Figure 16-3 offers a representation of processes and threads in a single-processor system.

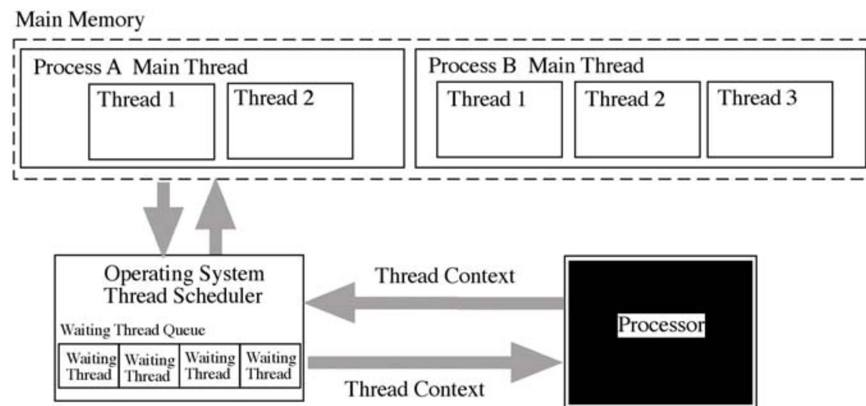


Figure 16-3: Processes and their Threads Executing in a Single-Processor Environment

Referring to Figure 16-3 — two processes A and B are executing. Process A contains three threads: Main, Thread 1 and Thread 2. Process B contains four threads: Main, Thread 1, Thread 2, and Thread 3. A thread is the smallest unit of code to which the operating system assigns processing time. A thread executes within the context of its containing or owning process and *application domain*.

As you can see in Figure 16-3, the operating system thread scheduler coordinates thread execution. Waiting threads sit in a thread queue until they are loaded into the processor. Each thread has a data structure known as a *thread context*. The thread context is a snapshot of the state of the processor and other execution details that must be preserved so that the thread can pick up execution where it left off when next loaded into the processor.

In a single-processor system, the operating system allocates processor time with a *time-slicing* scheme. Each thread gets a little bit of time to execute before being *preempted* by the next waiting thread, at which point, if it's not finished with its business, it takes its place in the thread queue to wait another turn at the processor. This diagram makes clear that in a single-processor system, the notion of concurrently executing applications is just an illusion pulled off by the operating system quickly switching threads in and out of the processor. Figure 16-4 shows how things might look on a multiprocessor system. Referring to Figure 16-4 — now we can really get some work done. In a multiprocessor system, two threads can actually execute concurrently, but the operating system still uses time-slicing to manage their execution and keep the whole show running smoothly.

Returning once again to my earlier vacation analogy, when you're on single-threaded vacation, the relax process does everything related to the vacation in one thread of execution. That's why you must stop relaxing and fetch yourself a drink and something to eat. When you're on multithreaded vacation, the relax process concentrates on relaxing and hands off the chores of drink and food fetching to separate threads, You come away from a multithreaded vacation feeling a lot more relaxed! (*Well, at least until you arrive at the airport anyway.*)

Vacation Gone Bad

There is a possibility, even on multithreaded vacation, for you to return home tense and frustrated. This can occur if the drink and food fetching threads misbehave. How might this happen? Assume for a moment, if you will, that you are not the only process on the beach. Laying next to you is a nasty little someone named "create-hate-and-discontent". He told his food and drink fetching threads they were special and gave them an order to cut in front of the line whenever possible. Your threads get booted from the bar and grill counters more frequently because of the higher priority of create-hate-and-discontent's threads. You suffer because your threads take longer to fetch food and drink. This is only one example of how ill-behaved threads can bring one or more processes to a halt.

 © 2008 Rick Miller — All Rights Reserved

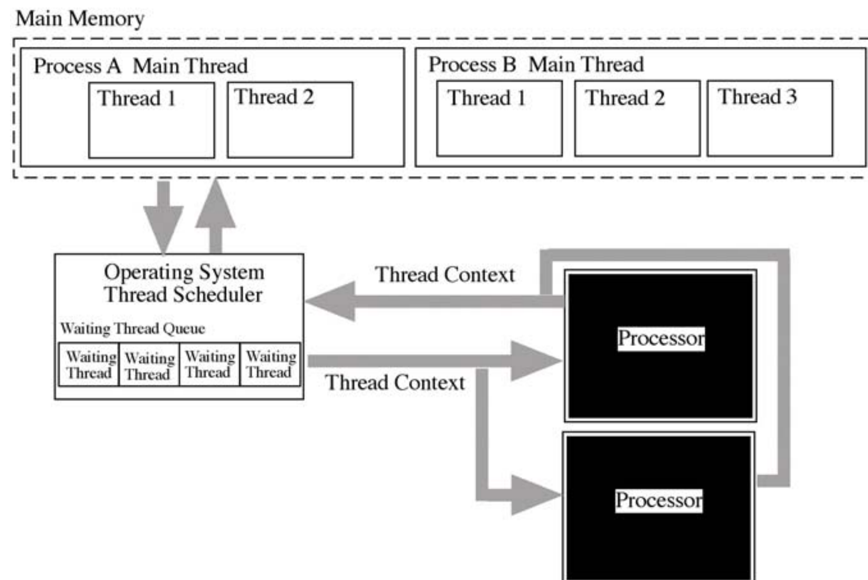


Figure 16-4: Processes and their Threads Executing in a Multiprocessor Environment

Quick Review

A process consists of one or more threads of execution, referred to simply as threads. A process always consists of at least one thread, the Main thread, which starts running when the process begins execution. A single-threaded process contains only one thread of execution. A multithreaded process contains more than one thread.

A thread is the smallest unit of code to which the operating system assigns processing time. A thread executes within the context of its containing or owning process and application domain.

Waiting threads sit in a thread queue until they are loaded into the processor. Each thread has a data structure known as a thread context. The thread context is a snapshot of the state of the processor and other execution details that must be preserved so that the thread can pick up execution where it left off when next loaded into the processor.

In a single-processor system the operating system allocates processor time with a time-slicing scheme. Each thread gets a little bit of time to execute before being preempted by the next waiting thread, at which point, if it's not finished with its business, it takes its place in the thread queue to wait another turn at the processor.

In a multiprocessor system, two threads can actually execute concurrently, but the operating system still uses time-slicing to manage their execution and keep the whole show running smoothly.

Creating Managed Threads With The Thread Class

In this section I will show you how to use the Thread class to create and manage the execution of threads in your programs. You'll find the Thread class, along with a whole lot of other useful stuff, in the System.Threading namespace. The Thread class allows you to create what are referred to as *managed threads*. They are called managed threads because you can directly manipulate each thread you create. You gain a lot of flexibility and power when you manage your own threads. However, with power and flexibility comes the responsibility of ensuring your threads behave well and properly handle exceptional conditions that may arise during their execution lifetime. This aspect of thread management gained increased importance in .NET 2.0 because, in most cases, unhandled exceptions lead to application termination.

The material in this section lays the foundation for the rest of the chapter. Once you understand the issues involved with creating and managing your own threads, you'll better understand why, in most cases, it's a good idea to let the runtime environment manage threads for you. However, before getting started, let's see just how little relaxing one does while on single-threaded vacation.

© 2008 Rick Miller — All Rights Reserved

Single-Threaded Vacation Example

How might the single-threaded vacation analogy be implemented in source code? Example 16.1 offers one possible solution.

16.1 SingleThreadedVacation.cs

```
1    using System;
2
3    public class SingleThreadedVacation {
4
5      private bool hungry;
6      private bool thirsty;
7
8      public SingleThreadedVacation(){
9        hungry = true;
10       thirsty = true;
11     }
12
13     public void FetchDrink(){
14       int steps_to_the_bar = 1000;
15       for(int i=0; i<steps_to_the_bar*2; i++){
16       if((i%100) == 0){
17         Console.WriteLine();
18         Console.Write("Fetching Drinks");
19       }else{
20         Console.Write(".");
21       }
22      }
23      Console.WriteLine();
24      thirsty = false;
25     }
26
27     public void FetchFood(){
28       int steps_to_the_grill = 1000;
29       for(int i=0; i<steps_to_the_grill*2; i++){
30       if((i%100)==0){
31         Console.WriteLine();
32         Console.Write("Fetching Food");
33       }else{
34         Console.Write(".");
35       }
36      }
37      Console.WriteLine();
38      hungry = false;
39     }
40
41     public static void Main(){
42       SingleThreadedVacation stv = new SingleThreadedVacation();
43       Console.WriteLine("Relaxing!");
44       while(stv.hungry && stv.thirsty){
45       stv.FetchDrink();
46       stv.FetchFood();
47       Console.WriteLine("Relaxing!");
48      }
49     }
50   }
```

Referring to Example 16.1 — the SingleThreadedVacation class contains two fields: hungry and thirsty, of type bool, which are initially set to true. It has two methods: FetchDrink() and FetchFood(). When each method is called, the `for` loop contained in each kills some time by "walking" the number of steps to the bar or grill and back again. Each method prints to the console a status message every 100 steps it takes.

The Main() method starting on line 41 starts by printing a message to the console saying it's "Relaxing!". It then enters the while loop where calls are made to FetchDrink() and FetchFood(). Since the whole program executes in a single thread of execution (*i.e.,* the Main() method's thread,) the FetchDrink() method must run to conclusion before the call to FetchFood() can be made. The FetchFood() method must then execute and return before the message "Relaxing!" can again be printed to the screen. Figure 16-5 shows SingleThreadedVacation in action.

Multithreaded Vacation Example

Let's now see how much more relaxing you can do on a multithreaded vacation. Example 16.2 gives the code for the MultiThreadedVacation class.

 © 2008 Rick Miller — All Rights Reserved

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\SingleThreadedVacation>singlethreadedvacation
Relaxing!

Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................

Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Relaxing!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\SingleThreadedVacation>
```

Figure 16-5: SingleThreadedVacation Program Output

16.2 MultiThreadedVacation.cs

```
1    using System;
2    using System.Threading;
3
4    public class MultiThreadedVacation {
5
6      private bool hungry;
7      private bool thirsty;
8
9      public MultiThreadedVacation(){
10       hungry = true;
11       thirsty = true;
12     }
13
14     public void FetchDrink(){
15       int steps_to_the_bar = 1000;
16       for(int i=0; i<steps_to_the_bar*2; i++){
17       if((i%100) == 0){
18         Console.WriteLine();
19         Console.Write("Fetching Drinks");
20       }else{
21         Console.Write(".");
22       }
23      }
24      Console.WriteLine();
25      thirsty = false;
26     }
27
28     public void FetchFood(){
29       int steps_to_the_grill = 1000;
30       for(int i=0; i<steps_to_the_grill*2; i++){
31       if((i%100)==0){
32         Console.WriteLine();
33         Console.Write("Fetching Food");
34       }else{
35         Console.Write(".");
36       }
37      }
38      Console.WriteLine();
```

 © 2008 Rick Miller — All Rights Reserved

```
39      hungry = false;
40     }
41
42     public static void Main(){
43       MultiThreadedVacation mtv = new MultiThreadedVacation();
44      Thread drinkFetcher = new Thread(mtv.FetchDrink);
45      Thread foodFetcher = new Thread(mtv.FetchFood);
46      Console.WriteLine("Relaxing!");
47
48      while(mtv.hungry && mtv.thirsty){
49        if(!drinkFetcher.IsAlive) drinkFetcher.Start();
50        if(!foodFetcher.IsAlive) foodFetcher.Start();
51        Console.Write("Relaxing!");
52      }
53     }
54   }
```

Referring to Example 16.2 — this code is structurally very similar to the previous example. The only changes made were to the insides of the Main() method where two thread objects are created on lines 44 and 45 named drinkFetcher and foodFetcher respectively. Note that to create a thread in this fashion, you supply to the Thread constructor the name of a method you want to execute in the separate thread. (Here the method signatures conform to the ThreadStart delegate signature.) The drinkFetcher thread executes the FetchDrink() method while the foodFetcher thread executes the FetchFood() method.

A check is made in the body of the `while` loop to see if each thread is alive, meaning "Has it been started?" If not, it is started by calling its Thread.Start() method. As soon as these threads are started, the Main() thread can go back to printing the message "Relaxing!" to the console. Figure 16-6 shows a partial listing of the MultiThreadedVacation program's output. As you'll note from looking at Figure 16-6 there's a lot more relaxing going on!

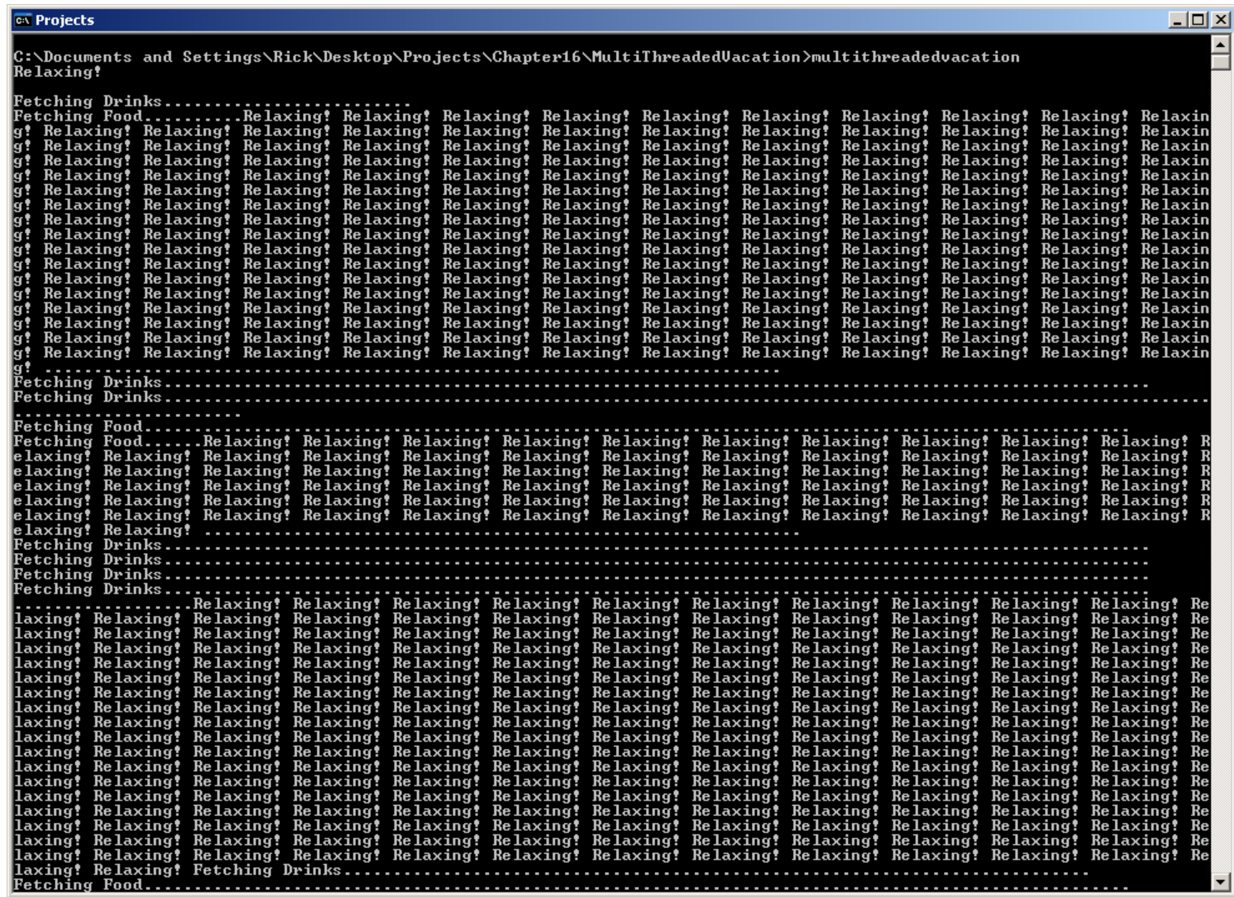


Figure 16-6: MultiThreadedVacation Program Output - Partial Listing

© 2008 Rick Miller — All Rights Reserved

Thread States

A thread can assume several different states during its execution lifetime, as shown in Figure 16-7.

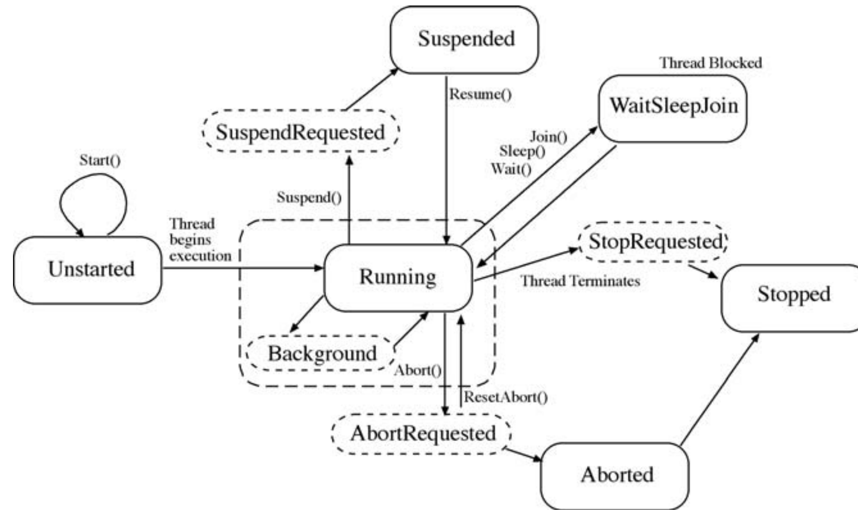


Figure 16-7: Thread States and Transition Initiators

Referring to Figure 16-7 — important points to note include the following: A call to a thread's Start() method does not immediately put the thread into the Running state. A call to Start() only notifies the operating system that the thread can now be started. Also, a thread can be in multiple states simultaneously. For example, a Running thread can also be a Background thread, or a Suspended thread can also be in the AbortRequested state.

It's tricky at best to personally manage multiple threads by directly manipulating their states. In fact, Microsoft recommends you don't do it because it's hard to tell precisely what state a thread is actually in, or more importantly, at what point in the code the thread is at when you attempt to move it from one state to another. It's usually never a good idea to call Abort() on an executing thread, especially if you didn't start the thread. Another thing to consider is that the Suspend() and Resume() methods are now obsolete.

So where does that leave you with regards to managing your own threads? Well, you can start a thread with the Start() method and block its operation with the Monitor.Wait(), Thread.Sleep() and Thread.Join() methods. You can change a foreground thread into a background thread by setting its IsBackground property to true. As it turns out, this amount of control is really all you need to write well-behaved, multithreaded code. The following sections discuss and demonstrate the use of the more helpful Thread properties and methods.

Creating And Starting Managed Threads

To create a managed thread, pass in to the Thread constructor either a *ThreadStart* delegate or a *ParameterizedThreadStart* delegate. The ParameterizedThreadStart delegate lets you pass an argument object when you call the thread's Start() method.

ThreadStart Delegate

The ThreadStart delegate specifies a method signature that returns `void` and takes no arguments. There are two ways to pass a ThreadStart delegate into the Thread constructor: the *longhand* way and the *shorthand* way. The longhand way entails explicitly creating a new ThreadStart delegate object as the following code fragment suggests.

```
Thread thread1 = new Thread(new ThreadStart(Run)); // longhand
```

The shorthand method of creating a thread entails just passing the name of the method to the Thread constructor and letting it figure out if what you supplied conforms to the ThreadStart delegate as the following code fragment demonstrates:

```
Thread thread2 = new Thread(Run); // shorthand
```

Example 16.3 demonstrates the longhand and shorthand way of creating threads.

© 2008 Rick Miller — All Rights Reserved

16.3 ThreadStartDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class ThreadStartDemo {
5
6      private const int COUNT = 200;
7
8      public static void Run(){
9        for(int i=0; i<COUNT; i++){
10        Console.Write(Thread.CurrentThread.Name);
11      }
12     }
13
14     public static void Main(){
15       Thread thread1 = new Thread(new ThreadStart(Run)); // longhand way
16       Thread thread2 = new Thread(Run); // shorthand way
17       thread1.Name = "1";
18       thread1.Start();
19       thread2.Name = "2";
20       thread2.Start();
21     }
22   }
```

Referring to Example 16.3 — two thread objects are created in the Main() method. The first, thread1, is created the longhand way by passing the name of the Run() method to the ThreadStart constructor. The second, thread2, is created the shorthand way by passing the name of the Run() method directly to the Thread constructor. Each thread's Name property is set before calling its Start() method. The name of the thread is printed to the console in the body of the Run() method. Note that in this example the Run() method is static, but it could just as well have been an instance method. Figure 16-8 shows the results of running this program.

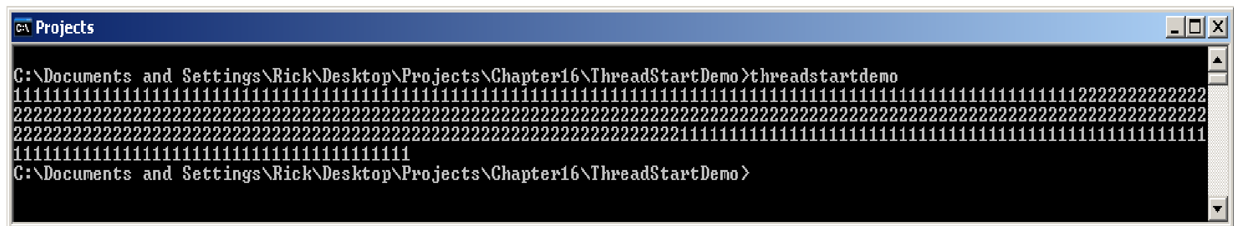


Figure 16-8: Results of Running Example 16.3

ParameterizedThreadStart Delegate: Passing Arguments To Threads

If you need to pass in an argument when you start a thread, your thread's execution method must conform to the ParameterizedThreadStart delegate signature. The ParameterizedThreadStart delegate method signature is shown in following code fragment:

```
public void MethodName(object obj)
```

Like its ThreadStart delegate cousin, you can create threads the longhand or shorthand way. Example 16.4 shows the ParameterizedThreadStart delegate in action.

16.4 ParameterizedThreadStartDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class ParameterizedThreadStartDemo {
5
6      private const int COUNT = 200;
7
8      public static void Run(object value){
9        for(int i=0; i<COUNT; i++){
10        Console.Write(value);
11      }
12     }
13
14     public static void Main(){
15       Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
16       Thread thread2 = new Thread(Run); // shorthand way
17       thread1.Start("Hello ");
18       thread2.Start("World! ");
19     }
20   }
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 16.4 — The static Run() method has been modified to conform to the ParameterizedThreadStart delegate method signature. In this case I am passing the parameter named “value” directly to the Console.Write() method, which will automatically call the Object.ToString() method. (**Note:** Here I’m only targeting the interface as specified by the Object class. If I expect some other type of object I must cast to the expected type.) Pass the argument to the thread when you call its Start() method, as is shown on lines 17 and 18. Figure 16-9 shows the results of running this program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ParameterizedThreadStart>parameterizedthreadstartdemo
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello World! World! World! World! World! World! World! World! World! World! World! W
orld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Wo
rld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Wor
ld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Worl
d! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World
! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World!
 World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World!
World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! W
orld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Wo
rld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Wor
ld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Worl
d! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World
! Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ParameterizedThreadStart>_
```

Figure 16-9: Results of Running Example 16.4

Blocking A Thread With Thread.Sleep()

If all goes well, a thread, once started, charges forward and executes until it completes its assigned task. If it can’t finish its assigned task in the allotted time slice, the operating system preempts the thread and swaps it out with another waiting thread. This swapping continues until the thread in question finishes its business or until something dreadful happens and it ends prematurely. Take a good look at Figure 16-9 and you’ll see how thread1 prints the message “Hello” over and over until it’s swapped out with thread2, which then starts to print “World!”.

In many situations, you’ll want a thread to do something and then take a short break to let other threads have a go at the processor. Example 16.5 adds a call to Thread.Sleep() to the body of the Run() method.

16.5 ParameterizedThreadStart.cs (With call to Sleep())

```
1    using System;
2    using System.Threading;
3
4    public class ParameterizedThreadStartDemo {
5
6      private const int COUNT = 200;
7
8      public static void Run(object value){
9        for(int i=0; i<COUNT; i++){
10        Console.Write(value);
11        Thread.Sleep(10);
12      }
13     }
14
15     public static void Main(){
16       Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
17      Thread thread2 = new Thread(Run); // shorthand way
18      thread1.Start("Hello ");
19      thread2.Start("World! ");
20     }
21   }
```

Referring to Example 16.5 — the call to Thread.Sleep() is made after the value is written to the console. Pass an integer argument to the Sleep() method indicating the time in milliseconds you want the thread to block. You can also pass in a TimeSpan object. Figure 16-10 shows the results of running this program. Note how different the output appears and how much slower the application seems to run because of the increased thread swapping that occurs.

 © 2008 Rick Miller — All Rights Reserved

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ParameterizedThreadStart>parameterizedthreadstartdemo
Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! Hel
lo World! World! Hello Hello World! Hello World! World! Hello Hello World! Hello World! World! Hello Hello World! World!
 Hello Hello World! World! Hello Hello World! Hello World! Hello World! World! Hello Hello World! Hello World! Hello Wor
ld! World! Hello World! Hello World! Hello Hello World! Hello World! World! Hello Hello World! Hello World! Hello World!
 World! Hello World! Hello World! Hello World! Hello Hello World! World! Hello Hello World! World! Hello Hello World! Wo
rld! Hello Hello World! World! Hello World! Hello World! Hello Hello World! World! Hello Hello World! World! Hello Hello
 World! World! Hello World! Hello World! Hello Hello World! Hello World! World! Hello World! Hello Hello World! Hello Wo
rld! Hello World! World! Hello Hello World! Hello World! Hello World! World! Hello Hello World! Hello World! Hello World
! World! Hello World! Hello Hello World! World! Hello World! Hello Hello World! Hello World! Hello World! World! Hello H
ello World! World! Hello Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hell
o World! Hello World! World! Hello Hello World! Hello World! Hello World! Hello World! World! Hello World! Hello World!
Hello Hello World! Hello World! World! Hello World! Hello Hello World! Hello World! Hello World! Hello World! Hello Worl
d! Hello World! Hello World! Hello World! World! Hello Hello World! World! Hello World! Hello Hello World! Hello World!
Hello World! Hello World! World! Hello Hello World! Hello World! Hello World! World! Hello Hello World! Hello World! Wor
ld! Hello Hello World! Hello World! World! Hello Hello World! Hello World! Hello World! World! Hello Hello World! Hello
World! World! Hello World! Hello World! Hello World! Hello Hello World! World! Hello Hello World! Hello World! Hello Wor
ld! Hello World! Hello World! Hello World! Hello World! World! Hello World! Hello Hello World! Hello World! World! Hello
 World! Hello World! Hello Hello World! World! Hello World! Hello Hello World! Hello World! Hello World! Hello World! Wo
rld! Hello Hello World! World! Hello World! Hello World! Hello World! Hello Hello World! World! Hello World! Hello Hello
 World! Hello World! Hello World! Hello World! Hello World! World! Hello World! Hello Hello World! Hello World! Hello Wo
rld! World! Hello Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World
! World! Hello Hello World! World! Hello Hello World! World! Hello World! Hello
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ParameterizedThreadStart>
```

Figure 16-10: Results of Running Example 16.5

Blocking A Thread With Thread.Join()

Another way to coordinate thread interaction is to explicitly block one thread until another thread completes execution. You can to this by calling the Thread.Join() method via the thread reference you want to yield to. For example, if you want the Main thread to block until thread2 completes execution, then in the Main thread you would call thread2.Join(). I want to show you two examples to demonstrate the use of the Join() method. The first, Example 16.6, builds on the previous example and adds a `for` loop in the Main() method that prints a message to the console. I've put the call to the thread2.Join() in the body of the `for` loop but it's commented out in this example.

16.6 JoinDemo.cs (Version 1)

```
1    using System;
2    using System.Threading;
3
4    public class JoinDemo {
5
6      private const int COUNT = 100;
7
8      public static void Run(object value){
9        for(int i=0; i<COUNT; i++){
10        Console.Write(value);
11        Thread.Sleep(10);
12      }
13     }
14
15     public static void Main(){
16       Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
17       Thread thread2 = new Thread(Run); // shorthand way
18       thread1.Start("Hello ");
19       thread2.Start("World! ");
20       for(int i = 0; i< 10; i++){
21         Console.Write("\n------- Main Thread Message --------");
22        //if(i==1) thread2.Join();
23      }
24     }
25   }
```

Referring to Example 16.6 — I've added a `for` loop to the end of the Main() method that loops ten times printing a message to the console. I've commented out line 22 for now so you can compare the output of this program with the output of the next example. Figure 16-11 shows the results of running this program. Referring to Figure 16-11 — note how thread1 and thread2 each print a message before sleeping. When the Main thread gets its chance to execute, it runs to completion.

Example 16.7 gives the JoinDemo program with line 22 in action. Figure 16-12 shows the results of running the program. Note the difference in the output between figures 16-11 and 16-12. The `for` loop in the Main thread makes it through two loops before being told to block until thread2 completes execution. (*i.e.,* thread2.Join())

 © 2008 Rick Miller — All Rights Reserved

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\JoinDemo>joindemo
Hello World!
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------Hello World! Hello World! World! Hello Hello World! World! Hello World! Hello World!
 Hello World! Hello World! Hello World! Hello Hello World! World! Hello Hello World! Hello World! Hello World! World! He
llo Hello World! Hello World! Hello World! World! Hello World! Hello World! Hello Hello World! World! Hello World! Hello
 World! Hello Hello World! World! Hello World! Hello World! Hello Hello World! World! Hello Hello World! World! Hello Wo
rld! Hello Hello World! World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! Hello World! World
! Hello World! Hello Hello World! World! Hello Hello World! World! Hello World! Hello World! Hello Hello World! Hello Wo
rld! Hello World! Hello World! World! Hello Hello World! Hello World! World! Hello Hello World! World! Hello World! Hell
o Hello World! World! Hello Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello H
ello World! World! Hello World! Hello World! Hello World! Hello Hello World! World! Hello World! Hello World! Hello Worl
d! Hello World! Hello Hello World! World! Hello Hello World! Hello World! Hello World! Hello World! World! Hello World!
Hello Hello World! Hello World! World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! World! Hel
lo
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\JoinDemo>_
```

Figure 16-11: Results of Running Example 16.6

16.7 JoinDemo.cs (Version 2)

```
1     using System;
2     using System.Threading;
3
4     public class JoinDemo {
5
6       private const int COUNT = 100;
7
8       public static void Run(object value){
9         for(int i=0; i<COUNT; i++){
10        Console.Write(value);
11        Thread.Sleep(10);
12       }
13      }
14
15      public static void Main(){
16        Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
17        Thread thread2 = new Thread(Run); // shorthand way
18        thread1.Start("Hello ");
19        thread2.Start("World! ");
20        for(int i = 0; i< 10; i++){
21          Console.Write("\n------- Main Thread Message --------");
22          if(i==1) thread2.Join(); // the Main thread will block on thread2 after second loop
23       }
24      }
25    }
```

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\JoinDemo>joindemo
Hello World!
------- Main Thread Message --------
------- Main Thread Message --------World! Hello World! Hello World! Hello Hello World! Hello World! Hello World! World!
 Hello World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! Hello World! Hello World! Hello Wor
ld! Hello World! Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello
 World! Hello Hello World! World! Hello Hello World! World! Hello Hello World! World! Hello Hello World! World! Hello Wo
rld! Hello Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello Hello World! World
! Hello Hello World! World! Hello Hello World! World! Hello Hello World! World! Hello World! Hello World! Hello World! H
ello World! Hello Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello Hello World
! World! Hello Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello W
orld! Hello Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello Hello World! World! Hello Worl
d! Hello World! Hello World! Hello World! Hello Hello World! World! Hello World! Hello World! Hello Hello World! World!
Hello Hello World! World! Hello Hello World! World! Hello Hello World! World! Hello Hello World! World! Hello Hello Worl
d!
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\JoinDemo>
```

Figure 16-12: Results of Running Example 16.7

© 2008 Rick Miller — All Rights Reserved

Foreground vs. Background Threads

A thread can be either a *foreground* thread or a *background* thread. The difference being that a foreground thread will keep the .NET runtime alive so long as it is running. A background thread, however, will be shutdown by the .NET runtime when it shuts down.

Managed threads are created as foreground threads. Example 16.8 gives an example of a foreground thread.

16.8 ForegroundThreadDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class ForegroundThreadDemo {
5
6       public static void Run(){
7         bool keepgoing = true;
8         while(keepgoing){
9           Console.Write("Please enter a letter or 'Q' to exit: ");
10          String s = Console.ReadLine();
11            switch(s[0]){
12              case 'Q': keepgoing = false;
13                        break;
14              default: break;
15           }
16         }
17      }
18
19     public static void Main(){
20       Thread thread1 = new Thread(Run);
21       thread1.Start();
22     }
23   }
```

Referring to Example 16.8 — the Main() method exits right after calling thread1.Start(). The Run() method loops continuously reading input from the console until the user enters the letter ‘Q’. Since thread1 is a foreground thread, it keeps the .NET runtime running as long as it’s executing. Figure 16-13 shows the results of running this program.

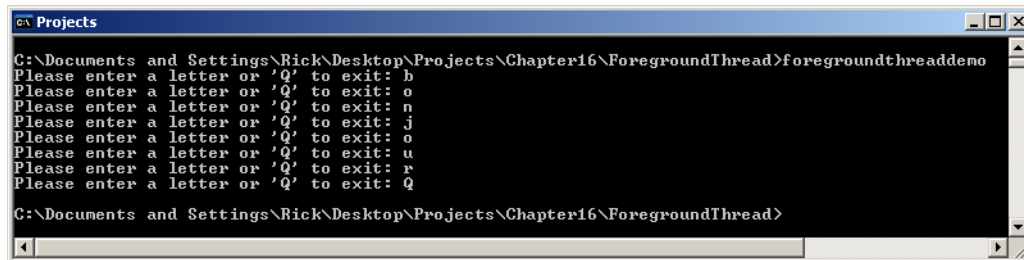


Figure 16-13: Results of Running Example 16.8

To change a foreground thread to a background thread, set the thread’s IsBackground property to true. Example 16.9 provides a slight modification to the previous example and makes thread1 a background thread.

16.9 BackgroundThreadDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class BackgroundThreadDemo {
5
6       public static void Run(){
7         bool keepgoing = true;
8         while(keepgoing){
9           Console.Write("Please enter a letter or 'Q' to exit: ");
10          String s = Console.ReadLine();
11          switch(s[0]){
12            case 'Q': keepgoing = false;
13                     break;
14            default: break;
15          }
16        }
17      }
18
19     public static void Main(){
20       Thread thread1 = new Thread(Run);
21       thread1.IsBackground = true;
22       thread1.Start();
23     }
24   }
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 16.9 — on line 21, thread1's IsBackground property is set to true. Its Start() method is called on the next line and the Main() method exits. Thus, thread1 is stopped along with the .NET runtime execution environment. Figure 16-14 shows the very brief results of running this program.

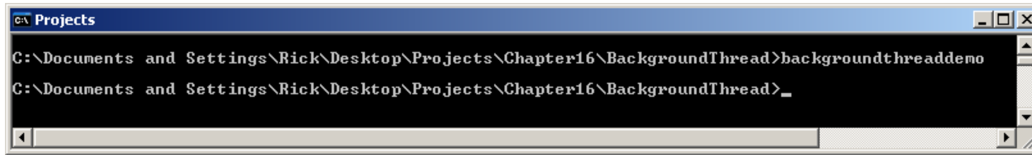


Figure 16-14: Results of Running Example 16.9

Quick Review

A thread can assume several different states during its execution lifetime. These states include: *Unstarted*, *Running*, *Background*, *SuspendRequested*, *Suspended*, *WaitSleepJoin*, *StopRequested*, *Stopped*, *AbortRequested*, and *Aborted*.

A call to a thread's Start() method does not immediately put the thread into the Running state. A call to Start() only notifies the operating system that the thread can now be started. Also, a thread can be in multiple states simultaneously. For example, a Running thread can also be a Background thread, or a Suspended thread can also be in the AbortRequested state.

It's tricky at best to personally manage multiple threads by directly manipulating their states. In fact, Microsoft recommends you don't do it because it's hard to tell precisely what state a thread is actually in or, more importantly, at what point in the code the thread is at when you attempt to move it from one state to another. It's usually never a good idea to call Abort() on an executing thread, especially if you didn't start the thread. Another thing to consider is that the Suspend() and Resume() methods are now obsolete.

To create a managed thread, pass to the Thread constructor either a ThreadStart delegate or a ParameterizedThreadStart delegate.

The ThreadStart delegate specifies a method signature that returns `void` and takes no arguments. There are two ways to pass the ThreadStart delegate to the Thread constructor: the *longhand* way and the *shorthand* way. The longhand way entails explicitly creating a new ThreadStart delegate object as the following code fragment suggests.

```
Thread thread1 = new Thread(new ThreadStart(Run)); // longhand
```

The shorthand method of creating a thread entails just passing the name of the method to the Thread constructor and letting it figure out if what you supplied conforms to the ThreadStart delegate as the following code fragment demonstrates:

```
Thread thread2 = new Thread(Run); // shorthand
```

If you need to pass in an argument when you start a thread, the thread's execution method must conform to the ParameterizedThreadStart delegate signature. The ParameterizedThreadStart delegate method signature is shown in following code fragment:

```
public void MethodName(object obj)
```

Like its ThreadStart delegate cousin, you can create threads the longhand or shorthand way. Pass the argument to the thread via its Start() method. Remember to cast the argument to the appropriate type in the body of the thread's execution method.

If all goes well, a thread, once started, charges forward and executes until it completes its assigned task. If it can't finish its assigned task in the allotted time slice, the operating system *preempts* the thread and swaps it out with another waiting thread. This swapping continues until the thread in question finishes its business or until something dreadful happens and it ends prematurely. Call the Thread.Sleep() method to force your thread to block and give other threads a chance to execute.

Another way to coordinate thread interaction is to explicitly block one thread until another thread completes execution. You can to this by calling the Thread.Join() method via the thread reference you want to yield to. For example, if you want the Main thread to block until thread2 completes execution, then in the Main thread you would call thread2.Join().

 © 2008 Rick Miller — All Rights Reserved

A thread can be either a *foreground* thread or a *background* thread. The difference being that a foreground thread will keep the .NET runtime alive so long as it is running. A background thread, however, will be shutdown by the .NET runtime when it shuts down.

Creating Threads With The BackgroundWorker Class

Background threads are especially helpful when used with GUI applications as they allow time-intensive activities to proceed while minimizing the impact to the user interface experience. The System.ComponentModel.*BackgroundWorker* class makes it easy and convenient to create background threads that do heavy lifting behind the scenes while relieving you of the burden of explicitly managing those threads.

The BackgroundWorker class provides this convenience and ease of use by allowing you to assign event handler methods to its various events. These events include *DoWork*, *ProgressChanged*, and *RunWorkerCompleted*. Example 16.10 shows the BackgroundWorker class in action. This program displays a small window with three buttons and three labels. When you click one of the buttons it fires the background worker to do that particular task. The tasks, in this case, are to simply print a short message to the console and update the color of the label when the task starts running and when it completes.

16.10 BackgroundWorkerDemo.cs

```
1     using System;
2     using System.Drawing;
3     using System.Threading;
4     using System.Windows.Forms;
5     using System.ComponentModel;
6
7     public class BackgroundWorkerDemo : Form {
8
9        private Button button1;
10       private Button button2;
11       private Button button3;
12       private Label label1;
13       private Label label2;
14       private Label label3;
15       private BackgroundWorker bw1;
16       private BackgroundWorker bw2;
17       private BackgroundWorker bw3;
18
19       public BackgroundWorkerDemo(){
20         InitializeComponents();
21       }
22
23       private void InitializeComponents(){
24         button1 = new Button();
25         button2 = new Button();
26         button3 = new Button();
27         label1 = new Label();
28         label2 = new Label();
29         label3 = new Label();
30         bw1 = new BackgroundWorker();
31         bw2 = new BackgroundWorker();
32         bw3 = new BackgroundWorker();
33
34         button1.Text = "Do Something";
35         button1.AutoSize = true;
36         button1.Click += ButtonOne_Click;
37         label1.BackColor = Color.Green;
38         bw1.DoWork += DoWorkOne;
39         bw1.RunWorkerCompleted += ResetLabelOne;
40
41         button2.Text = "Do Something Else";
42         button2.AutoSize = true;
43         button2.Click += ButtonTwo_Click;
44         label2.BackColor = Color.Green;
45         bw2.DoWork += DoWorkTwo;
46         bw2.RunWorkerCompleted += ResetLabelTwo;
47
48         button3.Text = "Do Something Different";
49         button3.AutoSize = true;
50         button3.Click += ButtonThree_Click;
51         label3.BackColor = Color.Green;
52         bw3.DoWork += DoWorkThree;
53         bw3.RunWorkerCompleted += ResetLabelThree;
```


© 2008 Rick Miller — All Rights Reserved

```
54
55        TableLayoutPanel tlp1 = new TableLayoutPanel();
56        tlp1.RowCount = 2;
57        tlp1.ColumnCount = 3;
58        tlp1.SuspendLayout();
59        this.SuspendLayout();
60        tlp1.AutoSize = true;
61        tlp1.Dock = DockStyle.Left;
62        tlp1.Controls.Add(button1);
63        tlp1.Controls.Add(button2);
64        tlp1.Controls.Add(button3);
65        tlp1.Controls.Add(label1);
66        tlp1.Controls.Add(label2);
67        tlp1.Controls.Add(label3);
68        this.Controls.Add(tlp1);
69        this.AutoSize = true;
70        this.AutoSizeMode = AutoSizeMode.GrowOnly;
71        this.Height = tlp1.Height;
72        tlp1.ResumeLayout();
73        this.ResumeLayout();
74      }
75
76      private void ButtonOne_Click(Object sender, EventArgs e){
77        if(!bw1.IsBusy){
78          bw1.RunWorkerAsync(((Button)sender).Text);
79        }
80      }
81
82      private void ButtonTwo_Click(Object sender, EventArgs e){
83        if(!bw2.IsBusy){
84          bw2.RunWorkerAsync(((Button)sender).Text);
85        }
86      }
87
88      private void ButtonThree_Click(Object sender, EventArgs e){
89        if(!bw3.IsBusy){
90          bw3.RunWorkerAsync(((Button)sender).Text);
91        }
92      }
93
94      private void DoWorkOne(Object sender, DoWorkEventArgs e){
95        label1.BackColor = Color.Black;
96        for(int i=0; i<30000; i++){
97          Console.Write(e.Argument + " ");
98        }
99      }
100
101      private void DoWorkTwo(Object sender, DoWorkEventArgs e){
102        label2.BackColor = Color.Black;
103          for(int i=0; i<30000; i++){
104            Console.Write(e.Argument + " ");
105        }
106      }
107
108      private void DoWorkThree(Object sender, DoWorkEventArgs e){
109        label3.BackColor = Color.Black;
110        for(int i=0; i<30000; i++){
111          Console.Write(e.Argument + " ");
112        }
113      }
114
115      private void ResetLabelOne(Object sender, RunWorkerCompletedEventArgs e){
116        label1.BackColor = Color.Green;
117      }
118
119      private void ResetLabelTwo(Object sender, RunWorkerCompletedEventArgs e){
120        label2.BackColor = Color.Green;
121      }
122
123      private void ResetLabelThree(Object sender, RunWorkerCompletedEventArgs e){
124        label3.BackColor = Color.Green;
125      }
126
127
128    [STAThread]
129    public static void Main(){
130      Application.Run(new BackgroundWorkerDemo());
131    }// end Main
132
133  } // end class definition
```

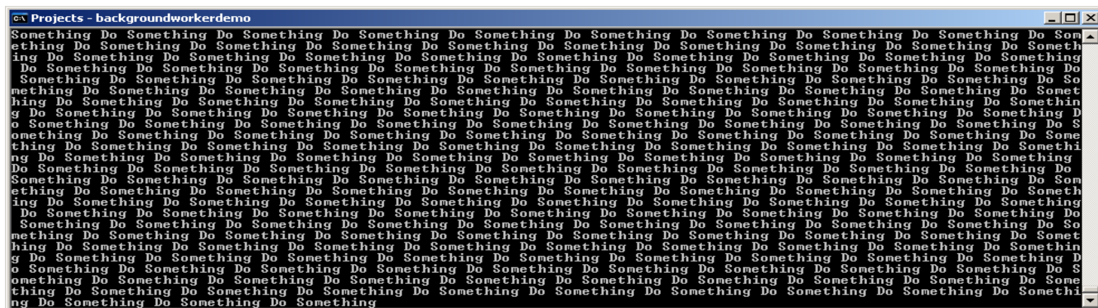

© 2008 Rick Miller — All Rights Reserved

Referring to Example 16.10 — in this code I create three buttons, three labels, and three BackgroundWorker objects named bw1, bw2, and bw3 respectively. To each background worker's *DoWork* event I assign a method that conforms to the *DoWorkEventHandler* delegate. These methods are named DoWorkOne(), DoWorkTwo(), and DoWorkThree(). To each background worker's *RunWorkerCompleted* event I assign a method that conforms to the *RunWorkerCompletedEventHandler* delegate. I named these methods ResetLabelOne(), ResetLabelTwo(), and ResetLabelThree().

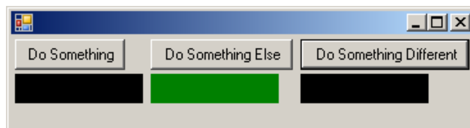
To each button's Click event I assign methods that conform to the EventHandler delegate. I've named these methods ButtonOne_Click(), ButtonTwo_Click(), and ButtonThree_Click(). A click on each button calls its assigned event handler method. The event handler method kicks off a background worker thread by calling its RunWorkAsync() method. In this case, I'm passing in to the call to the RunWorkAsync() method the text of the clicked button.

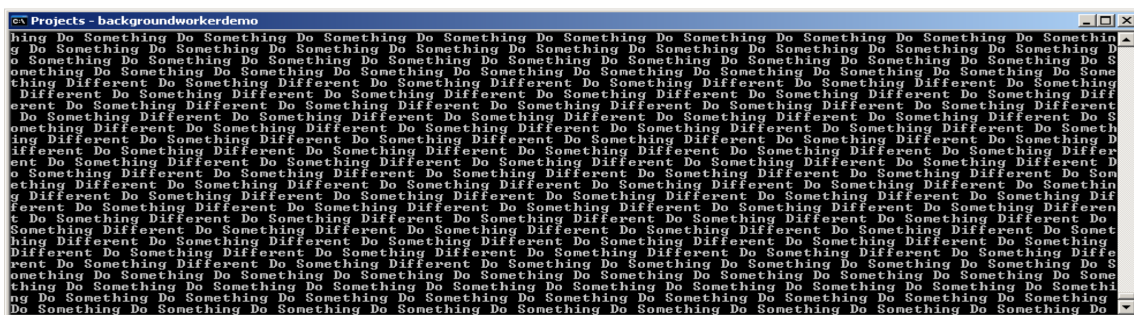
A call to a background worker's RunWorkAsync() method fires its DoWork event. Any DoWorkEventHandlers assigned to the background worker's DoWork event are then called. Before actually making the call to RunWorkerAsync(), I check to see if the background worker is busy by polling its IsBusy property. If the background worker is currently running an asynchronous operation, the IsBusy property returns true.

When the background worker thread completes, its RunWorkerCompleted event fires resulting in a call to any assigned RunWorkerCompletedEventHandler methods. Figure 16-15 shows the results of running this program.



Then click the Do Something Different button.





When both threads complete, each label's color is reset to green.

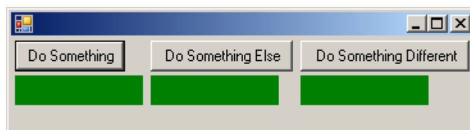


Figure 16-15: One Particular Result of Running Example 16.10

 © 2008 Rick Miller — All Rights Reserved

Quick Review

Background threads are especially helpful when used with GUI applications as they allow time-intensive activities to proceed while minimizing the impact to the user interface experience. The System.ComponentModel.*BackgroundWorker* class makes it easy and convenient to create background threads that do heavy lifting behind the scenes while relieving you of the burden of explicitly managing those threads. The BackgroundWorker class provides this convenience and ease of use by allowing you to assign event handler methods to its various events. These events include *DoWork*, *ProgressChanged*, and *RunWorkerCompleted*. A call to a BackgroundWorker's RunWorkAsync() method fires its DoWork event.

Thread Pools

The .NET runtime execution environment maintains and manages a *pool* of background threads for each application. You have access to these threads via the static methods of the ThreadPool class.

Beginning with .NET 2.0 Service Pack 1, each application's thread pool contains, by default, 250 worker threads per processor and 500 I/O completion port threads per processor. In this section, I will only show you how to use thread pool worker threads.

Important things to know about the application thread pool include the following:

- The ThreadPool class is static; its functionality is meant only to be used via its static methods.
- You can adjust the maximum number of threads in the pool via the ThreadPool.SetMaxThreads() method.
- To start a thread, pass the name of an execution method to the ThreadPool.QueueUserWorkItem() method.
- The thread pool contains a certain number of idle threads that are ready to execute. This number is adjusted via the ThreadPool.SetMinThreads() method. Too many idle threads extract a performance penalty because each idle thread requires stack space and other resources.
- The creation of new ThreadPool threads is throttled to one every 500 milliseconds. If you are spawning a large number of threads, you'll need to keep this throttling activity in mind.
- ThreadPool managed threads are background threads and will be terminated when your application exits.
- You have no control over a ThreadPool thread other than its initial creation.

Example 16.11 shows how easy it is to use ThreadPool threads. In this example, I spawn 45 separate threads with the help of the ThreadPool class. Following the creation of each thread, I print out the number of available threads.

16.11 ThreadPoolDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class ThreadPoolDemo {
5
6      private const int COUNT = 20000;
7
8      public static void Run(object stateInfo){
9        for(int i=0; i<COUNT; i++){
10       Console.Write(stateInfo + " ");
11       Thread.Sleep(100);
12      }
13     }
14
15     public static void Main(){
16        int workerThreads = 0;
17       int completionPortThreads = 0;
18       ThreadPool.GetMinThreads(out workerThreads, out completionPortThreads);
19       Console.WriteLine("Minimum number of worker threads in thread pool: {0} ", workerThreads);
20       Console.WriteLine("Minimum number of completion port threads in thread pool: {0} ",
21                         completionPortThreads);
22       ThreadPool.GetAvailableThreads(out workerThreads, out completionPortThreads);
23       Console.WriteLine("Available worker threads in thread pool: {0} ", workerThreads);
24       Console.WriteLine("Available completion port threads in thread pool: {0} ", completionPortThreads);
25
26       for(int i = 0; i<45; i++){
27         ThreadPool.QueueUserWorkItem(new WaitCallback(Run), i);
```


© 2008 Rick Miller — All Rights Reserved

```
28          Thread.Sleep(1000); // sleep twice as long as it takes to start a threadpool thread
29          ThreadPool.GetAvailableThreads(out workerThreads, out completionPortThreads);
30          Console.Write("\nAvailable worker threads in thread pool: {0} ", workerThreads);
31          Console.WriteLine("\nAvailable completion port threads in thread pool: {0}", completionPortThreads);
32        }
33      } // end Main() method
34    } // end class definition
```

Referring to Example 16.11 — each new thread is created in the body of the `for` loop that begins on line 26. Note on line 27 that the ThreadPool.QueueUserWorkItem() method requires a WaitCallBack object. I have supplied the name of the thread execution method to the WaitCallBack constructor and pass the resulting object as an argument to the QueueUserWorkItem() method. On line 28, I put the Main() method thread to sleep for twice as long as it takes to create a new ThreadPool thread, and then print the number of available threads to the console.

In this example, I have modified the signature of the Run() method to conform to the WaitCallBack delegate. This allows me to pass arguments to the Run() method when I kick off each thread with the QueueUserWorkItem() method.

Figure 16.16 shows a partial result of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ThreadPools>threadpooldemo
Minimum number of worker threads in thread pool: 2
Minimum number of completion port threads in thread pool: 2
Available worker threads in thread pool: 500
Available completion port threads in thread pool: 1000
0 0 0 0 0 0 0 0 0 0
Available worker threads in thread pool: 499
Available completion port threads in thread pool: 1000
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Available worker threads in thread pool: 498
Available completion port threads in thread pool: 1000
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Available worker threads in thread pool: 498
Available completion port threads in thread pool: 1000
2 0 1 2 0 1 2 0 1 2 0 1 2 0 3 1 2 3 0 1 2 0 3 1 2 0 3 1 2 0 3 1 2
Available worker threads in thread pool: 496
Available completion port threads in thread pool: 1000
0 3 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 4 2 3 0 4 1 2 3 0 1 4 2 0 3 4 1 2 0 3 1 4 2
Available worker threads in thread pool: 495
Available completion port threads in thread pool: 1000
3 0 4 1 2 3 0 4 1 2 3 0 4 1 2 3 0 4 1 2 0 3 1 4 5 2 0 3 4 1 5 2 0 3 4 1 5 2 0 3 4 1 5 2 0 3 1 4 5 2
Available worker threads in thread pool: 494
Available completion port threads in thread pool: 1000
3 0 4 1 5 2 3 0 4 1 5 2 0 3 1 4 5 2 0 3 4 1 5 2 3 0 1 4 5 2 6 0 3 1 4 5 2 6 3 0 4 1 5 2 6 3 0 1 4 5 2 6 3 0 4 1 5 2 6 3

Available worker threads in thread pool: 493
Available completion port threads in thread pool: 1000
```

Figure 16-16: Partial Result of Running Example 16.11

Quick Review

The .NET runtime execution environment maintains and manages a pool of background threads for each application. You have access to these threads via the static methods of the ThreadPool class. By default, each application's thread pool contains 250 worker threads per processor and 500 I/O completion port threads per processor. Pass the name of your thread execution method to the WaitCallBack constructor; pass the WaitCallBack object to the ThreadPool.QueueUserWorkItem() method.

Asynchronous Method Calls

Multithreading is built into the very core of the .NET runtime execution environment. You can call any method asynchronously with the help of a delegate. You can do this via a delegate's BeginInvoke() and EndInvoke() methods. Don't go looking for these methods in the System.Delegate documentation; the .NET runtime environment creates them automatically when you declare and define a new delegate type. The thread that executes an asynchronous method comes from the application thread pool and is therefore a background thread.

To make an asynchronous method call follow these steps:

- Create a new delegate type that specifies the method signature of your thread execution method.
- Create your thread execution method making sure its method signature matches that of the delegate you created in the first step.
- Create an instance of the delegate, passing the name of the thread execution method to its constructor.

 © 2008 Rick Miller — All Rights Reserved

- Call the BeginInvoke() method on the delegate object, supplying any necessary thread execution method arguments and two additional arguments of type AsyncCallback and an Object respectively. I will discuss the purpose of the AsyncCallback and Object parameters shortly.
- The call to BeginInvoke() returns an IAsyncResult object that can be used to query the state of the asynchronous method call's execution progress. The IAsyncResult.AsyncState property is a reference to the last object supplied in the call to the BeginInvoke() method.
- Do any required work in the calling method while the asynchronous method call executes.
- Call the EndInvoke() method to properly wrap-up the asynchronous method call and fetch the results.

Example 16.12 shows the asynchronous call mechanism in action.

16.12 AsynchronousCallDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class AsynchronousCallDemo {
5
6      private const int COUNT = 100;
7      public delegate void RunDelegate(String message);
8
9      public static void Run(String message){
10       for(int i=0; i<COUNT; i++){
11        Console.Write(message + " ");
12        Thread.Sleep(100);
13      }
14     }
15
16     public static void Main(){
17      RunDelegate runDelegate1 = new RunDelegate(Run);
18      RunDelegate runDelegate2 = new RunDelegate(Run);
19      IAsyncResult result1 = runDelegate1.BeginInvoke("Hello", null, null);
20      IAsyncResult result2 = runDelegate2.BeginInvoke("World!", null, null);
21      while(!result1.IsCompleted && !result2.IsCompleted){
22        Console.Write(" - ");
23        Thread.Sleep(1000);
24      }
25      runDelegate1.EndInvoke(result1);
26      runDelegate2.EndInvoke(result2);
27      Console.WriteLine("\nMain thread exiting now...bye!");
28     } // end Main() method
29   } // end class definition
```

Referring to Example 16.12 — on line 7, a new delegate type is declared named RunDelegate. The RunDelegate specifies a method that takes one String argument. The Run() method on line 9 conforms to the RunDelegate method signature specification. In the Main() method, I created two RunDelegate instances named runDelegate1 and runDelegate2. In the call to the RunDelegate constructor, I pass the name of the Run() method. I start the asynchronous methods by calling the BeginInvoke() method on each delegate instance passing in the required string argument and two null values representing the AsyncCallback and AsyncState objects, which are not being used in this case.

The `while` statement on line 21 loops until both IAsyncResult.IsCompleted properties are true. It prints the '-' character to the console and then sleeps for 1000 milliseconds to let the other two threads have a go at the processor.

On lines 25 and 26, the EndInvoke() method is called on each delegate instance, passing in the appropriate IAsyncResult reference. Figure 16-17 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>asynchronouscalldemo
 - Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! Hello World!
World! Hello  - Hello World! Hello World! World! Hello Hello World! World! Hello Hello World! Hello World! Hello World!
Hello World!  - Hello World! World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! World! Hello
World! Hello  - Hello World! World! Hello Hello World! Hello World! World! Hello World! Hello World! Hello Hello World!
World! Hello  - World! Hello World! Hello Hello World! World! Hello World! Hello World! Hello Hello World! World! Hello
Hello World!  - World! Hello World! Hello Hello World! World! Hello Hello World! Hello World! World! Hello World! Hello
World! Hello  - World! Hello Hello World! World! Hello World! Hello World! Hello World! Hello Hello World! Hello World!
Hello World! World! Hello  - Hello World! Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello
World! Hello Hello World!  - World! Hello Hello World! World! Hello World! Hello Hello World! Hello World! World! Hello
Hello World! Hello World!  - Hello World! World! Hello Hello World! World! Hello World! Hello Hello World! World! Hello
World! Hello Hello World!  - World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello
Hello World!
Main thread exiting now...bye!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>
```

Figure 16-17: Results of Running Example 16.12

© 2008 Rick Miller — All Rights Reserved

Obtaining Results From An Asynchronous Method Call

The are several ways to obtain results from an asynchronous method call. If the method returns a value, the call to the delegate's EndInvoke() method returns that value. If the method takes one or more `out` or `ref` parameters, these will be included in the EndInvoke() method's parameter list as well. (**Note:** Remember, a delegate's BeginInvoke() and EndInvoke() methods are automatically generated.) Example 16.13 demonstrates the use of the EndInvoke() method to retrieve an asynchronous method call's return value.

16.13 AsyncCallWithResultsDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class AsyncCallWithResultsDemo {
5
6      private const int COUNT = 100;
7      public delegate int SumDelegate(int a, int b);
8
9      public static int Sum(int a, int b){
10       return a + b;
11     }
12
13     public static void Main(){
14      SumDelegate sumDelegate1 = new SumDelegate(Sum);
15      SumDelegate sumDelegate2 = new SumDelegate(Sum);
16      IAsyncResult result1 = sumDelegate1.BeginInvoke(1, 2, null, null);
17      IAsyncResult result2 = sumDelegate2.BeginInvoke(3, 4, null, null);
18      while(!result1.IsCompleted && !result2.IsCompleted){
19        Thread.Sleep(100);
20      }
21      int sum1 = sumDelegate1.EndInvoke(result1);
22      int sum2 = sumDelegate2.EndInvoke(result2);
23      Console.WriteLine("The result of the first async method call is: {0}", sum1);
24      Console.WriteLine("The result of the second async method call is: {0}", sum2);
25      Console.WriteLine("\nMain thread exiting now...bye!");
26     } // end Main() method
27   } // end class definition
```

Referring to Example 16.13 — I defined a delegate on line 7 named SumDelegate that takes two integer arguments and returns an integer value. The Sum() method on line 9 conforms to the SumDelegate signature. In the Main() method, two SumDelegate objects are created named sumDelegate1 and sumDelegate2. The BeginInvoke() method is called on each delegate. Note how the multiple arguments are passed to the asynchronous method call. On line 18, the `while` loop spins until both method calls complete, which in this case doesn't take too long because of the simplicity of the Sum() method. On lines 21 and 22, the results of each method call are obtained via the call to each delegate's EndInvoke() method and the values written to the console. Figure 16-18 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>AsyncCallWithResultsDemo
The result of the first async method call is: 3
The result of the second async method call is: 7

Main thread exiting now...bye!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>
```

Figure 16-18: Results of Running Example 16.13

Providing A CallBack Method To BeginInvoke()

The BeginInvoke() method allows you to pass in a callback method that is automatically called when the asynchronous method completes execution. It also allows you to pass in an object argument to that callback method. Note that up until now I have been calling the BeginInvoke() method with the last two arguments set to null. *(i.e.,* sumDelegate1.BeginInvoke(1, 2, null, null)) To pass in a callback method, you'll need to write a method that conforms to the AsyncCallBack delegate method signature, which returns `void` and takes one argument of type IAsyncResult as the following code snippet shows:

```
void MethodName(IAsyncResult result)
```


© 2008 Rick Miller — All Rights Reserved

The IAsyncResult interface specifies an AsyncState property of type Object, meaning it can contain any type of object. You can pass in whatever your heart desires! To use this object in the callback method, you'll need to access the IAsyncResult.AsyncState property and cast it to the expected type. Example 16.14 demonstrates the use of a callback method.

16.14 AsyncCallWithCallBackDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class AsyncCallWithCallBackDemo {
5
6      private const int COUNT = 100;
7      public delegate int SumDelegate(int a, int b);
8
9      public static int Sum(int a, int b){
10       return a + b;
11     }
12
13     public static void WrapUp(IAsyncResult result){
14       SumDelegate sumDelegate = (SumDelegate)result.AsyncState;
15       int sum = sumDelegate.EndInvoke(result);
16       Console.WriteLine("The result is: {0} ", sum);
17     }
18
19     public static void Main(){
20       SumDelegate sumDelegate1 = new SumDelegate(Sum);
21       SumDelegate sumDelegate2 = new SumDelegate(Sum);
22       IAsyncResult result1 = sumDelegate1.BeginInvoke(1, 2, new AsyncCallback(WrapUp), sumDelegate1);
23       IAsyncResult result2 = sumDelegate2.BeginInvoke(3, 4, new AsyncCallback(WrapUp), sumDelegate2);
24       while(!result1.IsCompleted && (!result2.IsCompleted)){
25         Console.WriteLine(" - ");
26         Thread.Sleep(10);
27       }
28       Console.WriteLine("\nMain thread exiting now...bye!");
29     } // end Main() method
30   } // end class definition
```

Referring to Example 16.14 — I have added a method on line 13 named WrapUp() that conforms to the AsyncCallBack delegate method signature. In this example, I'm using the WrapUp() method to make the call to a SumDelegate's EndInvoke() method. To do this, I must pass in a reference to a SumDelegate, which I do as the last argument to each SumDelegate's BeginInvoke() method call shown on lines 22 and 23.

So, what's going on here? I'm executing two asynchronous method calls via two SumDelegate references. When each asynchronously executed method returns, the method supplied as the callback method is automatically called. It's a nice way to call and forget. However, since this is a simple console application, and the threads being created to execute the asynchronous method calls are thread pool background threads, the Main() method must hang on for a while and do some stuff, for if it exits right away, the background threads will be destroyed before they get a chance to execute. You generally don't have this problem when you're writing a GUI application. Figure 16-19 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>AsyncCallWithCallBackDemo
 -
The result is: 3
The result is: 7

Main thread exiting now...bye!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>
```

Figure 16-19: Results of Running Example 16.14

Quick Review

Multithreading is built into the very core of the .NET runtime execution environment. You can call any method *asynchronously* with the help of a delegate. You can do this via a delegate's BeginInvoke() and EndInvoke() methods. Don't go looking for these methods in the System.Delegate documentation; the .NET runtime environment creates them automatically when you declare and define a new delegate type. The thread that executes an asynchronous method comes from the application thread pool and is therefore a background thread.

© 2008 Rick Miller — All Rights Reserved

Summary

A *process* consists of one or more threads of execution, referred to simply as *threads*. A process always consists of at least one thread, the Main thread, which starts running when the process begins execution. A single-threaded process contains only one thread of execution. A multithreaded process contains more than one thread.

A *thread* is the smallest unit of code to which the operating system assigns processing time. A thread executes within the context of its containing or owning process and application domain.

Waiting threads sit in a *thread queue* until they are loaded into the processor. Each thread has a data structure known as a *thread context*. The thread context is a snapshot of the state of the processor and other execution details that must be preserved so that the thread can pick up execution where it left off when next loaded into the processor.

In a single-processor system the operating system allocates processor time with a *time-slicing* scheme. Each thread gets a little bit of time to execute before being *preempted* by the next waiting thread, at which point, if it's not finished with its business, it takes its place in the thread queue to wait another turn at the processor.

In a multiprocessor system, two threads can actually execute concurrently, but the operating system still uses time-slicing to manage their execution and keep the whole show running smoothly.

A thread can assume several different states during its execution lifetime. These states include: *Unstarted*, *Running*, *Background*, *SuspendRequested*, *Suspended*, *WaitSleepJoin*, *StopRequested*, *Stopped*, *AbortRequested*, and *Aborted*.

A call to a thread's Start() method does not immediately put the thread into the Running state. A call to Start() simply notifies the operating system that the thread can now be started. Also, a thread can be in multiple states simultaneously. For example, a Running thread can also be a Background thread, or a Suspended thread can also be in the AbortRequested state.

It's tricky at best to personally manage multiple threads by directly manipulating their states. In fact, Microsoft recommends you don't do it because it's hard to tell precisely what state a thread is actually in, or more importantly, at what point in the code the thread is at when you attempt to move it from one state to another. It's usually never a good idea to call Abort() on an executing thread, especially if you didn't start the thread. Another thing to consider is that the Suspend() and Resume() methods are obsolete.

To create a managed thread, pass to the Thread constructor either a ThreadStart delegate or a ParameterizedThreadStart delegate.

The ThreadStart delegate specifies a method signature that returns `void` and takes no arguments. There are two ways to pass the ThreadStart delegate to the Thread constructor: the *longhand* way and the *shorthand* way. The longhand way entails explicitly creating a new ThreadStart delegate object as the following code fragment suggests.

```
Thread thread1 = new Thread(new ThreadStart(Run)); // longhand
```

The shorthand method of creating a thread entails just passing the name of the method to the Thread constructor and letting it figure out if what you supplied conforms to the ThreadStart delegate as the following code fragment demonstrates:

```
Thread thread2 = new Thread(Run); // shorthand
```

If you need to pass in an argument when you start a thread, the thread's execution method must conform to the ParameterizedThreadStart delegate signature. The ParameterizedThreadStart delegate method signature is shown in following code fragment:

```
public void MethodName(object obj)
```

Like its ThreadStart delegate cousin, you can create threads the longhand or shorthand way. Pass the argument to the thread via its Start() method. Remember to cast the argument to the appropriate type in the body of the thread's execution method.

If all goes well, a thread, once started, charges forward and executes until it completes its assigned task. If it can't finish its assigned task in the allotted time slice, the operating system preempts the thread and swaps it out with another waiting thread. This swapping continues until the thread in question finishes its business or until something dreadful happens and it ends prematurely. Call the Thread.Sleep() method to force a thread to *block* and give other threads a chance to execute.

Another way to coordinate thread interaction is to explicitly block one thread until another thread completes execution. You can to this by calling the Thread.Join() method via the thread reference you want to yield to. For example,

 © 2008 Rick Miller — All Rights Reserved

if you want the Main thread to block until thread2 completes execution then in the Main thread you would call thread2.Join().

A thread can be either a *foreground* thread or a *background* thread. The difference being that a foreground thread keeps the .NET runtime alive so long as it is running. A background thread, however, will be shutdown by the .NET runtime when it shuts down.

Background threads are especially helpful when used with GUI applications as they allow time-intensive activities to proceed while minimizing the impact to the user interface experience. The System.ComponentModel.BackgroundWorker class makes it easy and convenient to create background threads that do heavy lifting behind the scenes while relieving you of the burden of explicitly managing those threads. The BackgroundWorker class provides this convenience and ease of use by allowing you to assign event handler methods to its various events. These events include *DoWork*, *ProgressChanged*, and *RunWorkerCompleted*. A call to a BackgroundWorker's RunWorkAsync() method fires its DoWork event.

The .NET runtime execution environment maintains and manages a *pool* of background threads for each application. You have access to these threads via the static methods of the ThreadPool class. By default, each application's thread pool contains 250 worker threads per processor and 500 I/O completion port threads per processor. Pass the name of the thread execution method to the WaitCallBack constructor; pass the WaitCallBack object to the ThreadPool.QueueUserWorkItem() method.

Multithreading is built into the very core of the .NET runtime execution environment. You can call any method *asynchronously* with the help of a *delegate*. You can do this via a delegate's BeginInvoke() and EndInvoke() methods. Don't go looking for these methods in the System.Delegate documentation; the .NET runtime environment creates them automatically when you declare and define a new delegate type. The thread that executes an asynchronous method comes from the application thread pool and is therefore a background thread.

Skill-Building Exercises

1. **API Drill:** Explore the System.Threading namespace. List each class and describe its purpose.

2. **API Drill:** Explore the .NET Framework documentation and search for information about thread synchronization mechanisms. Pay particular attention to the *Interlocked* class, the *lock* keyword, the *Monitor* class, the *Mutex* class, and the *Semaphore* class. Describe in your own words how each synchronization mechanism works.

4. **Web Research:** Search the web for information about *deadlock* and *race* conditions. Briefly explain how each of the thread synchronization mechanisms you learned about in the previous exercise can be used to avoid either deadlock or race conditions.

5. **Web Research:** Procure and read the paper titled *Race Conditions: A Case Study* by Steve Carr, et. al.

6. **Programming Exercise:** Compile and run the sample programs presented in this chapter. Experiment by making various modifications to the programs and note the results of their execution.

7. **API Drill:** Explore the System.Collections.Generic namespace. Study each class and note how to use it in a multithreaded program.

Suggested Projects

1. **Multithreaded Water Tank:** Revisit the Automated Water Tank program given in Chapter 13, Examples 13.6 through 13.11, and make it a multithreaded program.

© 2008 Rick Miller — All Rights Reserved

Self-Test Questions

1. (True/False) An application always has at least one thread.

2. What happens to background threads when an application exits?

3. What's the difference between a background thread and a foreground thread?

4. (True/False) A managed thread immediately starts running when you call its Start() method.

5. Which two Thread methods are considered obsolete?

6. (True/False) It's generally considered a good idea to try to manage multiple threads by manipulating their states.

7. Beginning with .NET 2.0 Service Pack 1, how many ThreadPool worker threads are available per processor?

8. What's the difference between a ThreadStart delegate and a ParameterizedThreadStart delegate?

9. Which two delegate methods are used together to run methods asynchronously?

10. What's the difference between a process and a thread?

11. How does the typical operating system coordinate thread execution?

12. What term is used to describe what happens when one thread is removed from the processor in favor of another?

References

Atul Gupta, *How Many Threads Have I Got?*, [http://infosysblogs.com/microsoft/2007/04/how_many_threads_have_i_got.html]

Steve Carr, et. al, *Race Conditions: A Case Study*

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

© 2008 Rick Miller — All Rights Reserved

Notes

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Chapter 17

Contax T / Kodak Tri-X

Champs Elysées – Paris

File I/O

Learning Objectives

- Create and manipulate directories and files
- Create and manipulate text files
- Make a class serializable by using the Serializable attribute
- Serialize/deserialize objects to/from disk with the BinaryFormatter class
- Serialize/deserialize objects to/from disk with the XMLSerializer class
- Append data to existing files
- Properly handle file I/O exceptions
- List and describe the contents of the System.IO namespace
- Create and read log files using classes from the System.IO.Log namespace
- Use FileDialogs to graphically locate and open/save files

© 2008 Rick Miller — All Rights Reserved

Introduction

All but the most trivial software applications must preserve their data in some form or another. This chapter shows you how to preserve your application data to local files. These files might be located on a hard drive, a floppy disk, a USB drive, or some other type of media connected to your computer. In most cases, the type of media is of no concern to you because the operating system, and the storage device's driver software, handle the machine-specific details. All you need to know to conduct file Input/Output (I/O) operations is a handful of .NET Framework classes. The operating system does the rest.

You're going to learn a lot of cool things in this chapter, like how to manipulate files and directories, how to serialize and deserialize objects to disk, how to read and write text files, how to perform random access file I/O, how to write log files, and finally, how to use an OpenFileDialog to locate and open files. You will be surprised to learn you can do all these things with only a small handful of classes, structures, and enumerations, most of which are found in the System.IO namespace.

When you finish this chapter, you will have reached an important milestone in your C# programming career — you will be able to write applications that save data to disk. You will find this to be a critical skill to have in your programmer's toolbox.

Manipulating Directories And Files

In most all cases, data generated by an application and stored on an auxiliary storage device such as a hard disk, is saved as an organized, related collection of data in a structure commonly referred to as a *file*. I say "in most cases" because it is entirely possible to write data to an absolute or random position on a device, depending of course on what type of storage medium you're talking about. (*i.e.,* A disk drive works differently than a tape drive.)

It is the operating system's responsibility to manage the organization, reading, and writing of files. When you add a new storage device to your computer, it must first be formatted in a way that allows the operating system to access its data. The file management services provided by the operating system are part of a set of layered services that make it possible to build complex computing systems, as Figure 17-1 partially illustrates.

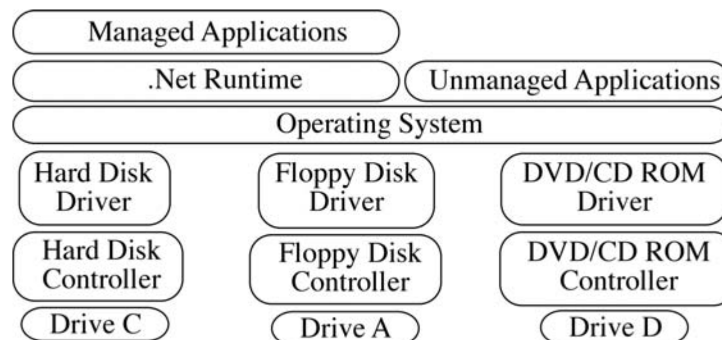


Figure 17-1: Simplified View of Service Layers

Referring to Figure 17-1 — attached storage devices interact with the operating system via an associated software interface referred to as a *driver*. Each device will have its own particular software driver that must be installed and recognized by the operating system before it will work. This applies not only to storage devices but to network cards, display devices, printers, etc. The operating system dictates the rules by which attached storage devices must play, and it is the responsibility of the storage device manufacturer to implement these rules in the device driver.

The operating system makes the services offered by its various device drivers available to running applications. Well-behaved applications target the operating system and do not directly interact with attached storage devices. (**Note:** .NET applications target the .NET runtime environment.)

© 2008 Rick Miller — All Rights Reserved

Files, Directories, And Paths

The Microsoft Windows operating system assigns each attached storage device a letter. On computers with only one hard drive, the letter assigned is 'C' and is referred to as your "C drive". If you have a 3.5 inch floppy drive, its assigned letter is 'A'. The operating system assigns the next available letter to the next available storage device. Thus, if you also have a CD/ROM or DVD drive, its letter will most likely be 'D'. If you plug in a removable USB drive, the operating system will assign to it the letter 'E' for as long as it's attached to the machine.

The file, from the operating system's point of view, is the fundamental storage organizational element. An application's associated data can be stored in one or more files. A file is located in another organizational element called a *directory*. A directory is a special type of file that contains a list of files and directories. A directory contained inside another directory is called a subdirectory. In modern operating systems like Windows or Apple's OS X, the metaphors *folder* and *subfolder* are used to refer to a directory and a subdirectory respectively.

The topmost directory structure on a storage device is referred to as the *root* directory. A particular drive's root directory is indicated by the name of the drive followed by a colon ':', followed by a backward slash character '\'. The root directory of the C drive would be "C:\". Figure 17-2 illustrates these concepts.

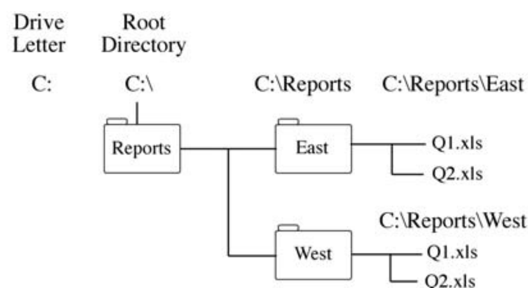


Figure 17-2: Typical Directory Structure

The location of a particular file within a directory structure is indicated by a string of characters called a *path*. The path to the file's location can be *absolute* or *relative*. An absolute path includes the name or letter of the drive and all directory and subdirectory names required to pinpoint the file's location. For example, referring to Figure 17-2 — the absolute path to the Microsoft Excel spreadsheet file named Q2.xls located in the East directory, which is located in the Reports directory, which is located in the root directory of the C drive would be:

"`C:\Reports\East\Q2.xls`".

Figure 17-3 illustrates the concept of an absolute path.

A relative path is the path to a file from some arbitrary starting point, usually a working directory.

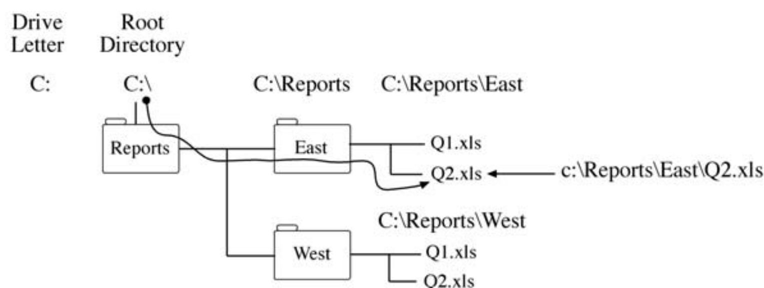


Figure 17-3: The Absolute Path to the Reports\East\Q2.xls File

Manipulating Directories And Files

You can easily create and manipulate directories and files with the help of several classes provided by the .NET Framework System.IO namespace. These include the *Path*, *File*, *FileInfo*, *Directory*, *DirectoryInfo*, and *DriveInfo* classes. The difference between the Directory/File classes vs. DirectoryInfo/FileInfo classes is that the former are static classes while the latter are non-static, meaning you can create instances of FileInfo and DirectoryInfo. Use the

© 2008 Rick Miller — All Rights Reserved

static class versions when you need to perform one or two operations on a directory or file. If you need to do more robust directory or file processing use the -Info versions.

The use of these classes is fairly straightforward. Example 17.1 offers a short program that prints out information about the current directory, the files it contains, and the drives available on the computer.

17.1 DirectoryClassDemo.cs

```
1    using System;
2    using System.IO;
3
4    public class DirectoryClassDemo {
5      public static void Main(){
6        Console.WriteLine("The full path name of the current directory is...");
7        Console.WriteLine("\t" + Directory.GetCurrentDirectory());
8         Console.WriteLine("The current directory has the following files...");
9         String[] files = Directory.GetFiles(Directory.GetCurrentDirectory());
10        foreach(String s in files){
11           FileInfo file = new FileInfo(s);
12           Console.WriteLine("\t" + file.Name);
13        }
14        Console.WriteLine("The computer has the following attached drives...");
15        String[] drives = Directory.GetLogicalDrives();
16        foreach(String s in drives){
17         Console.WriteLine("\t" + s);
18        }
19     }
20   }
```

Referring to Example 17.1 — this example actually demonstrates the use of both the static Directory class and the non-static FileInfo class. On line 7, the Directory.GetCurrentDirectory() method is used to get the absolute path to the current, or working, directory. (*i.e.,* The directory in which the program executes.) On line 9, the Directory.GetFiles() method returns an array of strings representing each of the files in the current working directory. (**Note:** The Directory.GetFileSystemEntries() method would return a string array with the names of all files and directories in the current working directory.)

Given the array of filename strings, the `foreach` statement on line 10 iterates over each entry, creates a new FileInfo object for each filename, and prints its name in the console. You could have simply printed out the array of strings, but that would give you the complete path name of each file. The FileInfo.Name property only returns the name of the file, not its complete path name.

Finally, on line 15, the Directory.GetLogicalDrives() method returns a string array containing the names of all drives connected to the computer. Figure 17-4 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\Directory>directoryclassdemo
The full path name of the current directory is...
        C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\Directory
The current directory has the following files...
        DirectoryClassDemo.cs
        DirectoryClassDemo.exe
The computer has the following attached drives...
        A:\
        C:\
        D:\
        E:\

C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\Directory>
```

Figure 17-4: Results of Running Example 17.1

Verbatim String Literals

From now on, you will find it more convenient to use *verbatim string literals* rather than ordinary strings when formulating path names. When using ordinary strings, you must precede special characters with the escape character ‘\’. For example, a path name formulated as an ordinary string would look like this:

```
String path = ”c:\\Reports\\East\\Q1.xls”; //ordinary string
```

Verbatim strings are formulated by preceding the string with the ‘@’ character, which signals the compiler to “...interpret the following string literally, including special characters and line breaks.” The path string given above would look like this as a verbatim string:

```
String path = @”c:\Reports\East\Q1.xls”; // verbatim string
```

 © 2008 Rick Miller — All Rights Reserved

Quick Review

In most all cases, data generated by an application and stored on an auxiliary storage device such as a hard disk, is saved as an organized, related collection of data in a structure commonly referred to as a *file*.

It is the operating system's responsibility to manage the organization, reading, and writing of files. When you add a new storage device to your computer, it must first be formatted in a way that allows the operating system to access its data.

The file, from the operating system's point of view, is the fundamental storage organizational element. An application's associated data can be stored in one or more files. A file is located in another organizational element called a *directory*. A directory is a special type of file that contains a list of files and directories. A directory contained inside another directory is called a *subdirectory*.

The topmost directory structure is referred to as the root directory. The root directory of a particular drive is indicated by the name of the drive followed by a colon ':', followed by a backward slash character '\'. The root directory of the C drive would be "C:\".

The location of a particular file within a directory structure is indicated by a string of characters called a *path*. The path to the file's location can be *absolute* or *relative*. An *absolute path* includes the name or letter of the drive and all directory and subdirectory names required to pinpoint the file's location. A *relative path* is the path to a file from some arbitrary starting point, usually a working directory.

You can easily create and manipulate directories and files with the help of several classes provided in the .NET Framework System.IO namespace. These include the *Path*, *File*, *FileInfo*, *Directory*, *DirectoryInfo*, and *DriveInfo* classes.

Verbatim strings are formulated by preceding the string with the '@' character which signals the compiler to "...interpret the following string literally, including special characters and line breaks."

Serializing Objects To Disk

The easiest way to save data to a file is via *serialization*. Serialization is the term used to describe the process of encoding objects in such a way as to facilitate their transmission out of the computer and into or onto some other type of media. Objects can be serialized to disk and then later *deserialized* and reconstituted into objects. The same objects can be serialized for transmission across a network and deserialized at the other end.

While powerful and convenient for you the programmer, serialization is the least flexible way to store data to disk because doing so ties you to the .NET platform. You can't edit the resulting data file. Well, you could edit the file, but because object information is encoded, it's not an ordinary text file, so it's highly likely that you'd screw something up if you did try to edit the file with, say, an ordinary text editor. One way around this is to serialize objects into an XML file.

The nice thing about serialization is that you can serialize single objects, or collections of objects. In this section I will show you how to serialize collections of objects using ordinary serialization with the help of the BinaryFormatter class, and XML serialization with the help of the XMLSerializer class.

Serializable Attribute

Before any object can be serialized it must be tagged as being serializable. You do this by tagging the class with the Serializable attribute. When dealing with collections of objects, not only must the collection itself be serializable — all the objects contained within the collection must be serializable as well. However, you need not worry about collections, and this includes arrays, as they are already tagged as being serializable. Example 17.2 demonstrates the use of the Serializable attribute to make the Dog class serializable.

17.2 Dog.cs

```
1    using System;
2
3    [Serializable]
4    public class Dog {
5
6       private String name = null;
7       private DateTime birthday;
```

 © 2008 Rick Miller — All Rights Reserved

```
8
9        public Dog(String name, DateTime birthday){
10            this.name = name;
11            this.birthday = birthday;
12       }
13
14       public Dog():this("Dog Joe", new DateTime(2005,01,01)){ }
15
16       public Dog(String name):this(name, new DateTime(2005,01,01)){ }
17
18
19      public int Age {
20        get {
21          int years = DateTime.Now.Year - _birthday.Year;
22          int adjustment = 0;
23            if(DateTime.Now.Month < _birthday.Month){
24              adjustment = 1;
25            }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
26                      adjustment = 1;
27                  }
28          return years - adjustment;
29        }
30      }
31
32       public DateTime Birthday {
33        get { return birthday; }
34        set { birthday = value; }
35
36       }
37
38       public String Name {
39        get { return name; }
40        set { name = value; }
41       }
42
43
44       public override String ToString(){
45         return (name + "," + Age);
46       }
47
48   } // end class definition
```

Referring to Example 17.2 — the Serializable attribute appears on line 3 just above the start of the class definition in square brackets. That's it! This tells the compiler that instances of the Dog class can be serialized. In the next section I'll show you how to serialize an array of Dog objects with the help of the BinaryFormatter class.

Serializing Objects With BinaryFormatter

To serialize an object to disk, you'll need to perform the following steps:

Step 1: Create a FileStream object with the name of the file you want to create on disk.

Step 2: Create a BinaryFormatter object and call its Serialize() method, passing in a reference to a FileStream object and a reference to the object you want to serialize.

Deserialization is the opposite of serialization. Deserialization is the process of reconstituting an object that has been previously serialized and turning it back into an object. To deserialize an object from disk, you must perform the following steps:

Step 1: Create a FileStream object that opens the file that contains the object you want to deserialize.

Step 2: Create a BinaryFormatter object and call its Deserialize() method passing in a reference to the FileStream object.

Step 3: The BinaryFormatter.Deserialize() method returns an object. This object must be cast to the appropriate type.

Example 17.3 offers a short program that serializes and deserializes an array of Dog objects. This program depends on the Dog class presented in Example 17.2.

17.3 MainApp.cs

```
1    using System;
2    using System.IO;
3    using System.Runtime.Serialization.Formatters.Binary;
4    using System.Runtime.Serialization;
5
6    public class MainApp {
7     public static void Main(String[] args){
```

 © 2008 Rick Miller — All Rights Reserved

```
8       /**********************************************
9         Create an array of Dogs and populate
10      **********************************************/
11      Dog[] dog_array = new Dog[3];
12
13      dog_array[0] = new Dog("Rick Miller", new DateTime(1965, 07, 08));
14      dog_array[1] = new Dog("Coralie Powell", new DateTime(1973, 08, 10));
15      dog_array[2] = new Dog("Kyle Miller", new DateTime(1990, 05, 01));
16
17      /*********************************************
18        Iterate over the dog_array and print values
19      *********************************************/
20      Console.WriteLine("-----Original Dog Array Contents---------------------");
21      for(int i = 0; i<dog_array.Length; i++){
22        Console.WriteLine(dog_array[i].Name + ", " + dog_array[i].Age);
23      }
24
25      /***********************************************
26        Serialize the array of dog objects to a file
27      ***********************************************/
28      FileStream fs = null;
29      try{
30        fs = new FileStream("DogFile.dat", FileMode.Create);
31        BinaryFormatter bf = new BinaryFormatter();
32        bf.Serialize(fs, dog_array);
33
34      }catch(IOException e){
35        Console.WriteLine(e.Message);
36      }catch(SerializationException se){
37        Console.WriteLine(se.Message);
38      }finally{
39         fs.Close();
40      }
41
42      /***********************************************
43        Deserialize the array of dogs and print values
44      ************************************************/
45         fs = null;                          //start fresh
46         Dog[] another_dog_array = null;     //here too!
47         try{
48           fs = new FileStream("DogFile.dat", FileMode.Open);
49           BinaryFormatter bf = new BinaryFormatter();
50           another_dog_array = (Dog[])bf.Deserialize(fs);
51           Console.WriteLine("-----After Serialization and Deserialization---------");
52           for(int i = 0; i<another_dog_array.Length; i++){
53                Console.WriteLine(another_dog_array[i].Name + ", " + another_dog_array[i].Age);
54           }
55
56         }catch(IOException e){
57
58         Console.WriteLine(e.Message);
59            }catch(SerializationException se){
60              Console.WriteLine(se.Message);
61            }finally{
62               fs.Close();
63       }
64    } // end Main() definition
65   } // end MainApp class definition
```

Referring to Example 17.3 — note the namespaces you must use to serialize objects to disk with a BinaryFormatter. These include System.IO, System.Runtime.Serialization, and System.Runtime.Serialization.Formatters.Binary. The first thing the program does is create an array of Dogs on line 11 and populate it with references to three Dog objects. The `for` loop starting on line 21 iterates over the dog_array and prints each dog's name and age to the console. The serialization process starts on line 28 with the declaration of the FileStream reference named fs. In the body of the `try` block that begins on line 29, the FileStream object is created using the filename "DogFile.dat" and a FileMode of Create. (**Note:** You can name your files anything you like within the rules of the operating system.)

The BinaryFormatter is created on line 31 and on the next line the Serialize() method is called passing in the reference to the FileStream (fs) and the reference to the array of dogs (dog_array). The appropriate exceptions are handled should something go wrong.

The deserialization process begins on line 45 by setting the reference fs to null and creating a completely new array to house the deserialized array of Dog objects. On line 48, a new FileStream object is created given the appropriate file name and a FileMode of Open. A new BinaryFormatter object is created on the following line and its Deserialize() method is called passing in a reference to the FileStream object. Note how the deserialized object is cast to an

 © 2008 Rick Miller — All Rights Reserved

array of Dogs (*i.e. Dog[]*). The `for` loop on line 52 iterates over another_dog_array and prints each dog's name and age to the console. Figure 17-5 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\SerializedDogs>mainapp
-----Original Dog Array Contents---------------------
Rick Miller, 42
Coralie Powell, 34
Kyle Miller, 17
-----After Serialization and Deserialization---------
Rick Miller, 42
Coralie Powell, 34
Kyle Miller, 17

C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\SerializedDogs>_
```

Figure 17-5: Results of Running Example 17.3

Serializing Objects With XMLSerializer

You can serialize objects to disk in XML format with the help of the XMLSerializer class. The steps required to serialize objects to an XML file are similar to those of ordinary serialization:

Step 1: Create a StreamWriter object passing in the name of the file where you want to save the object.

Step 2: Create an XMLSerializer object and call its Serialize() method passing in a reference to the file and to the object you want to serialize.

To deserialize an XML file you would do the following:

Step 1: Create a FileStream object passing in the name of the file you want to read.

Step 2: Create an XMLSerializer object and call its Deserialize() method.

Step 3: The Deserialize() method returns an object. You must cast this object to the appropriate type.

Example 17.4 gives a modified version of MainApp.cs that serializes an array of Dog objects to disk in an XML file.

17.4 MainApp.cs (Mod 1)

```
1    using System;
2    using System.IO;
3    using System.Xml;
4    using System.Xml.Serialization;
5
6    public class MainApp {
7     public static void Main(String[] args){
8       /************************************************
9         Create an array of Dogs and populate
10      ***********************************************/
11      Dog[] dog_array = new Dog[3];
12
13      dog_array[0] = new Dog("Rick Miller", new DateTime(1965, 07, 08));
14      dog_array[1] = new Dog("Coralie Powell", new DateTime(1973, 08, 10));
15      dog_array[2] = new Dog("Kyle Miller", new DateTime(1990, 05, 01));
16
17      /***********************************************
18        Iterate over the dog_array and print values
19      **********************************************/
20      Console.WriteLine("-----Original Dog Array Contents---------------------");
21      for(int i = 0; i<dog_array.Length; i++){
22        Console.WriteLine(dog_array[i].Name + ", " + dog_array[i].Age);
23      }
24
25      /************************************************
26        Serialize the array of dog objects to a file
27      ***********************************************/
28      TextWriter writer = null;
29      try{
30        writer = new StreamWriter("dogfile.xml");
31        XmlSerializer serializer = new XmlSerializer(typeof(Dog[]));
32        serializer.Serialize(writer, dog_array);
33
34
35      }catch(IOException ioe){
36        Console.WriteLine(ioe.Message);
```


© 2008 Rick Miller — All Rights Reserved

```
37      }catch(Exception ex){
38        Console.WriteLine(ex.Message);
39      }finally{
40         writer.Close();
41       }
42
43     /***********************************************
44        Deserialize the array of dogs and print values
45     ***********************************************/
46         FileStream fs = null;                          //start fresh
47         Dog[] another_dog_array = null;     //here too!
48         try{
49           fs = new FileStream("dogfile.xml", FileMode.Open);
50           XmlSerializer serializer = new XmlSerializer(typeof(Dog[]));
51           another_dog_array = (Dog[])serializer.Deserialize(fs);
52           Console.WriteLine("-----After Serialization and Deserialization---------");
53           for(int i = 0; i<another_dog_array.Length; i++){
54          Console.WriteLine(another_dog_array[i].Name + ", " + another_dog_array[i].Age);
55           }
56
57         }catch(IOException ioe){
58
59         Console.WriteLine(ioe.Message);
60            }catch(Exception ex){
61              Console.WriteLine(ex.Message);
62            }finally{
63               fs.Close();
64       }
65    } // end Main() definition
66   } // end MainApp class definition
```

Referring to Example 17.4 — note now that the namespaces required to serialize objects to an XML file include System.IO, System.XML, and System.XML.Serialization. The serialization process begins on line 28 with the declaration of a TextWriter reference. In the body of the try block, a StreamWriter object is actually created passing in the name of the file that will be used to hold the serialized dog_array. On line 31, an XMLSerializer object is created. Note that what gets passed as an argument to the constructor is the type of object that will be serialized. The Serialize() method is called on the following line passing in the reference to the output file (writer) and the object to be serialized (dog_array).

The deserialization process starts on line 46 with the declaration of the FileStream reference fs. Another dog array is declared named another_dog_array. In the body of the `try` block starting on line 48, the FileStream object is created passing in the name of the input file and a FileMode of Open. Next, an XMLSerializer object is created again passing to its constructor the type of object that will be deserialized. Lastly, the Deserialize() method is called passing in the name of the input file. The resulting object must be cast to the type Array of Dog (Dog[]). The `for` loop then iterates over the contents of another_dog_array and prints the name and age of each dog to the console. Figure 17-6 gives the results of running this program.

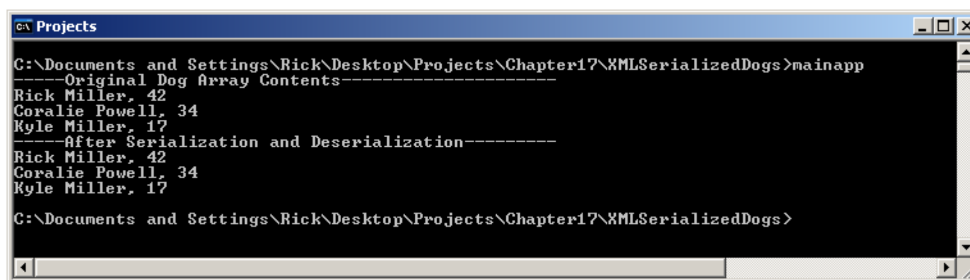


Figure 17-6: Results of Running Example 17.4

At this point you'll find it interesting to explore the contents of both the DogFile.dat and the dogfile.xml files. The DogFile.dat file appears to contain a log of gibberish, while the XML file is a readable text file that contains XML tags corresponding to the object or objects that were serialized. Example 17.5 gives the listing of dogfile.xml.

17.5 Contents of dogfile.xml

```
1    <?xml version="1.0" encoding="utf-8"?>
2    <ArrayOfDog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3                               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
4      <Dog>
5        <Birthday>1965-07-08T00:00:00</Birthday>
6        <Name>Rick Miller</Name>
7      </Dog>
```

 © 2008 Rick Miller — All Rights Reserved

```
8      <Dog>
9        <Birthday>1973-08-10T00:00:00</Birthday>
10       <Name>Coralie Powell</Name>
11     </Dog>
12     <Dog>
13       <Birthday>1990-05-01T00:00:00</Birthday>
14       <Name>Kyle Miller</Name>
15     </Dog>
16   </ArrayOfDog>
```

Quick Review

Object *serialization* provides an easy, convenient way for you to persist application data to disk. Object serialization is also the least flexible way to store application data because you can't edit the resulting file. Use a FileStream object and a BinaryFormatter to serialize objects to disk. Before an object can be serialized it must be tagged as being serializable with the Serializable attribute. Place the Serializable attribute above the class declaration line.

When serializing a collection of objects, remember that all objects contained within the collection must be serializable. You don't have to worry about the collections themselves, including ordinary arrays, as they are already tagged as being serializable.

You can get around the limitation of ordinary serialization by serializing objects to disk in XML format. Use the StreamWriter and XMLSerializer classes to serialize objects to disk in XML format. Use a FileStream and XMLSerializer to deserialize objects from an XML file.

Working With Text Files

One of the best ways to store data in a way that can be easily shared between different applications or different computer platforms is in a *text file*. The System.IO namespace provides two classes that make it easy to process text files: *StreamReader* and *StreamWriter*. The StreamReader class extends the abstract *TextReader* class; the StreamWriter extends the abstract *TextWriter* class.

Some Issues You Must Consider

Before you start writing code to process text files, you'll need to spend some time in the design phase working on exactly what format the text within your text file will have. By format I mean how the text is organized within the file. The decisions you make regarding this issue will vary according to your application's data storage needs. For example, a small database application might store records as separate lines of text. These lines may be, and usually are, separated by special characters referred to as *carriage-return/line-feed* (\r\n). Individual fields within each record may be further separated or *delimited* with another type of character. One character that's commonly used to delimit fields is the comma ','.

Another critically important point to consider is, "What data needs to be preserved in the text file?" For example, if you are working with Person objects within your program, and you want to save this data to a file, what data about each Person object must you save to allow the creation of Person objects later when the data is read from the file?

Also, how might the data be treated later in its life? Will it be read by another program? If so, what type of application is it and how will the data's format affect the application's performance.

Saving Dog Data To A Text File

Example 17.6 offers a short program that saves the data for an array of Dog objects to a text file. After the file is written, the program reads and parses the text file and recreates the array of Dog objects.

17.6 TextFileDemo.cs

```
1    using System;
2    using System.IO;
3
4    public class TextFileDemo {
5      public static void Main(){
6        /***********************************************
```


© 2008 Rick Miller — All Rights Reserved

```
7           Create an array of Dogs and populate
8       *********************************************/
9       Dog[] dog_array = new Dog[3];
10
11      dog_array[0] = new Dog("Rick Miller", new DateTime(1965, 07, 08));
12      dog_array[1] = new Dog("Coralie Powell", new DateTime(1973, 08, 10));
13      dog_array[2] = new Dog("Kyle Miller", new DateTime(1990, 05, 01));
14
15      /*********************************************
16        Iterate over the dog_array and print values
17      *********************************************/
18      Console.WriteLine("-----Original Dog Array Contents---------------------");
19      foreach(Dog d in dog_array){
20        Console.WriteLine(d.Name + ", " + d.Age);
21      }
22
23      /*******************************************
24        Save data to textfile
25      *******************************************/
26      TextWriter writer = null;
27      try{
28        writer = new StreamWriter("dogfile.txt");
29        foreach(Dog d in dog_array){
30         writer.WriteLine(d.Name + "," + d.Birthday.Year + "-" + d.Birthday.Month + "-" + d.Birthday.Day);
31        }
32        writer.Flush();
33      }catch(Exception e){
34        Console.WriteLine(e);
35      }finally{
36        writer.Close();
37      }
38
39      /*********************************************
40        Read data from text file and create objects...
41      *********************************************/
42      TextReader reader = null;
43      Dog[] another_dog_array = new Dog[3];
44      try{
45        reader = new StreamReader("dogfile.txt");
46        String s = String.Empty;
47        int count = 0;
48        while((s = reader.ReadLine()) != null){
49          String[] line = s.Split(',');
50          String name = line[0];
51          String[] dob = line[1].Split('-');
52          another_dog_array[count++] = new Dog(name, new DateTime(Int32.Parse(dob[0]), Int32.Parse(dob[1]),
53                                                                             Int32.Parse(dob[2])));
54        }
55      }catch(Exception e){
56        Console.WriteLine(e);
57      }finally{
58        reader.Close();
59      }
60
61       Console.WriteLine("-----------After writing to and reading from text file------------");
62       foreach(Dog d in another_dog_array){
63          Console.WriteLine(d.Name + ", " + d.Age);
64       }
65
66     } // end Main()
67   } // end class definition
```

Referring to Example 17-6 — the array of Dog reference is created as before and each dog's name and age is printed to the console. The start of the text file save process begins on line 26 with the declaration of the TextWriter reference named writer. In the body of the `try` block, a new StreamWriter is created passing in the name of the file in which to save the Dog object data. (dogfile.txt) The `foreach` loop iterates over each element of the array and calls the writer.WriteLine() method to write each dog's name and birthday information to disk. Note that in this case I am separating the name field from the birthday field with a comma.

To create a DateTime object later when I read the file, I will need to have the year, month, and day of the dog's birthday. I delimit each piece of the birthday with a hyphen '-'. When I have finished writing all the lines, I call the writer.Flush() method to actually write the data to disk.

The file read process begins on line 42 with the declaration of a TextReader reference. In the body of the `try` block, I create a StreamReader object passing in the name of the text file to read. I then process the text file according to the following algorithm:

- Declare a string variable in which will be stored each line as it is read from the text file.

© 2008 Rick Miller — All Rights Reserved

- Declare a count variable to control the process loop.
- Read the next line of the file and if it's not null, process the line like so:
  - Declare a string array to hold the individual fields of the string when it is split.
  - Call the String.Split() method to split the line into tokens based on the field delimiter ','.
  - Create a string variable called "name" and assign to it the first token of the split string.
  - Create another string array named dob (short for *date of birth*) to hold the split date field.
  - Call the String.Split() method on the second line token (i.e., line[1]) to split the dob.
  - Create the Dog object using the extracted fields.

As you can see, there is considerably more work involved with manipulating lines of text files. Figure 17-7 gives the results of running this program. Example 17.7 shows the contents of the dogfile.txt file.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\TextFileDemo>textfiledemo
-----Original Dog Array Contents----------------------
Rick Miller, 42
Coralie Powell, 34
Kyle Miller, 17
-----------After writing to and reading from text file------------
Rick Miller, 42
Coralie Powell, 34
Kyle Miller, 17

C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\TextFileDemo>_
```

Figure 17-7: Results of Running Example 17.6

17.7 Contents of dogfile.txt

```
1    Rick Miller,1965-7-8
2    Coralie Powell,1973-8-10
3    Kyle Miller,1990-5-1
```

Quick Review

The StreamReader and StreamWriter classes let you read and write text files. Text files are usually processed line-by-line. Lines of text are terminated with the special characters *carriage-return and line-feed* (\r\n). Each line can contain one or more fields *delimited* by some character. The comma ',' is a commonly used field delimiter. Individual fields can be further delimited as required.

Look to the objects in your program to determine the type of information your text file(s) must contain. You'll need to save enough data to recreate objects.

Process a text file by reading each line and breaking it into tokens with the String.Split() method. If one or more fields are also delimited, use the String.Split() method to tokenize the data as required.

Working With Binary Data

You can read and write binary data to a file with the help of the *BinaryReader* and *BinaryWriter* classes. The BinaryWriter class provides an overloaded Write() method that is used to write each of the simple types including strings and arrays of bytes and characters. The BinaryReader class provides an assortment of Read*Typename*() methods where *Typename* may be any one of the simple types to include strings and arrays of bytes and characters.

Example 17.8 shows the BinaryWriter and BinaryReader classes in action.

17.8 BinaryDataDemo.cs

```
1    using System;
2    using System.IO;
3
4    public class BinaryDataDemo {
5      public static void Main(){
6
7         int record_count = 5;
8         int record_number = 0;
9         int int_val = 125;
10        double double_val = -4567.00;
11        String string_val = "I love C#!";
12        bool bool_val = true;
13
```

 © 2008 Rick Miller — All Rights Reserved

```
14      /*********************************************************
15         Create the file and write the data with a BinaryWriter
16       *******************************************************/
17       BinaryWriter writer = null;
18        try{
19         writer = new BinaryWriter(File.Open("binaryfile.dat", FileMode.Create));
20         writer.Write(record_count);
21         for(int i=0; i<record_count; i++){
22           writer.Write(++record_number);
23           writer.Write(int_val);
24           writer.Write(double_val);
25           writer.Write(string_val);
26           writer.Write(bool_val);
27         }
28
29        }catch(Exception e){
30          Console.WriteLine(e);
31        }finally{
32          writer.Close();
33        }
34
35        /*******************************************************
36            Open the file and read the data with a BinaryReader
37          *******************************************************/
38       BinaryReader reader = null;
39       record_count = 0; // reset record count
40       try{
41         reader = new BinaryReader(File.Open("binaryfile.dat", FileMode.Open));
42         record_count = reader.ReadInt32();
43         for(int i=0; i<record_count; i++){
44           Console.WriteLine("Record #:     " + reader.ReadInt32());
45           Console.WriteLine("Int value:    " + reader.ReadInt32());
46           Console.WriteLine("Double value: " + reader.ReadDouble());
47           Console.WriteLine("String value: " + reader.ReadString());
48           Console.WriteLine("Bool value:   " + reader.ReadBoolean());
49           Console.WriteLine("-------------------------------------------------------");
50         }
51
52       }catch(Exception e){
53         Console.WriteLine(e);
54       }finally{
55         reader.Close();
56       }
57    } // end Main()
58  } // end class definition
```

Referring to Example 17.8 — on lines 7 through 12 I declare a set of variables of various different types. I use the variable named record_count to indicate the number of records I'll be writing to and reading from the file. The variable named record_number is incremented for each record that is written to the file and will thus be different for each record. The rest of the variables remain unchanged for the duration of the program.

The BinaryWriter reference named writer is declared on line 17 and is used to write the various simple-type variable values to a file named binaryfile.dat. The `for` loop starting on line 21 writes five records to the file. In this case the boundary of each record, or set of binary values, is demarcated only by the combined length of data written to the file during each iteration of the `for` loop. Also, in this case, the combined length of data written to the file with each iteration of the `for` loop is constant because I don't modify the length of the string variable. If I did, then you'd have variable length records.

The BinaryReader reference named reader is declared on line 38 and is used to read the binary values from the file. How does the reader object know where to read? This is where the concept of a *file position pointer* comes into play. The file position pointer is a variable within the reader object that keeps track of the start of the next read location. It is advanced to the next location based on the length of the type that was just read. For example, if you read an integer value, the file position pointer is advanced 4 bytes. If the next value read is a string, the pointer is advanced to a point equal to the length of the string. That's why it's important to know exactly what type you are reading and where in the file you are reading it from. In the case of Example 17.8 above, the `for` loop starting on line 43 simply reads the values from the file in the order in which they were written. Figure 17-8 shows the results of running this program.

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\Binary>binarydatademo
Record #:      1
Int value:     125
Double value: -4567
String value: I love C#!
Bool value:   True
------------------------------------------------------------
Record #:      2
Int value:     125
Double value: -4567
String value: I love C#!
Bool value:   True
------------------------------------------------------------
Record #:      3
Int value:     125
Double value: -4567
String value: I love C#!
Bool value:   True
------------------------------------------------------------
Record #:      4
Int value:     125
Double value: -4567
String value: I love C#!
Bool value:   True
------------------------------------------------------------
Record #:      5
Int value:     125
Double value: -4567
String value: I love C#!
Bool value:   True
------------------------------------------------------------

C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\Binary>
```

Figure 17-8: Results of Running Example 17.8

Quick Review

Use the BinaryReader and BinaryWriter classes to read and write binary data to disk. The BinaryWriter class provides an overloaded Write() method that is used to write each of the simple types including strings and arrays of bytes and characters. The BinaryReader class provides an assortment of Read*Typename*() methods where *Typename* may be any one of the simple types to include strings and arrays of bytes and characters.

Random Access File I/O

You can conduct *random access file operations* with the help of the *BinaryReader*, *BinaryWriter*, and *FileStream* classes. The FileStream class provides a Seek() method that allows you to position the file pointer at any point within a file. As you learned in the previous section, the BinaryReader and BinaryWriter classes provide methods for reading and writing binary, string, byte, and character array data.

There are many ways to go about random access file operations, but generally speaking, you must know a little something about how data is organized in a file so that you know where to find what you are looking for. When seeking a specific record location, you must know where one record ends and another begins. This is not the same as reading lines of text where line terminators provide clues as to where one line ends and a new one begins. In most random access file situations, record length is fixed. (*i.e.,* fixed-length records) A fixed-length record can contain a mixture of binary and character data, but each field within the record is a known size. Seeking the location of a particular record within the file requires the setting of the file position pointer value to a multiple of the record length. The number of records a file contains can be calculated by dividing the file length in bytes by the record length in bytes. You could, of course, randomly seek to any position in a file, but who knows what data you will find there!

In this section I'm going to show you a rather extended example of random access file operations. The example code and resulting application provides a solution to the legacy datafile adapter project specification given in Figure 17-9. Please take some time now to review the project specification before proceeding to the next section.

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

 © 2008 Rick Miller — All Rights Reserved

schema definition and compare it to 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?

```
                              Legacy Datafile Adapter
                               Project Specification

Objectives:
  - Demonstrate your ability to conduct random access file I/O operations using the BinaryReader, Binary-
    Writer, and FileStream classes
  - Demonstrate your ability to implement a non-trivial interface
  - Demonstrate your ability to translate low-level exceptions into higher-level, user-defined, applica-
    tion-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 .NET technology. A first step in this initiative is to create
    C# adapters to existing legacy data stores. Given an interface definition, example legacy data file,
    and legacy data file schema definition, write a C# class that serves as an adapter object to a legacy
    data file.

Given:
  - C# 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 17-9: Legacy Datafile Adapter Project Specification

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 BinaryReader class has a method named ReadInt16(). The method name derives from the System.Int16 structure that represents the short data type in the .NET Framework. A short is a two-byte value. The ReadInt16() 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 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 17-10 shows a simple analysis performed with a spreadsheet.

© 2008 Rick Miller — All Rights Reserved

	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 17-10: Header and Record Length Analysis

Referring to Figure 17-10 — 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 you write the adapter.

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 that a 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.) An object can be used as a synchronization point by using the C# `lock` keyword or the Monitor.Enter() and Monitor.Exit() methods. 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 attribute to the entire method in question (UpdateRecord() and DeleteRecord()) or 2) control access only to the critical section of code within each method. Within the locked 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 by calling the Monitor.Wait() method. The Wait() method blocks the current thread and adds it 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 and is ready to release the lock, it can notify other waiting threads to wake up by calling the Monitor.Pulse() method. I have used the `lock` keyword along with Monitor.Wait() and Monitor.Pulse() methods to synchronize access to critical code sections within the DatafileAdapter class.

 © 2008 Rick Miller — All Rights Reserved

Monitor.Enter()/Monitor.Exit() vs. The lock Keyword

The `lock` keyword is equivalent to the Monitor.Enter()/Monitor.Exit() method combination. You certainly could use the Monitor.Enter()/Monitor.Exit() combination to control access to a critical code section, but you must take measures to ensure the Monitor.Exit() method gets called at some point. To do this, Microsoft recommends that you use them within the body of a `try/finally` block. The `lock` keyword automatically wraps the Monitor.Enter() and Monitor.Exit() methods in a `try/finally` block for you. Figure 17-11 shows you how the use of the Monitor.Enter()/Monitor.Exit() methods compares to the use of the `lock` keyword.

```
try {                                      lock(Object){
    Monitor.Enter(Object);

    // critical code section                  // critical code section


   } catch(ArgumentNullException e){       }
       // handle  appropriately
   } finally {
         Monitor.Exit(Object);
     }
```

Figure 17-11: Monitor.Enter()/Monitor.Exit() vs. the lock Keyword

Translating Low-Level Exceptions Into Higher-Level Exception Abstractions

The System.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, extend the Exception class and add any customized behavior required. (Exceptions are discussed in detail in Chapter 15.) In your adapter code, you catch and handle the low-level exception when it occurs, 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 FileStream, BinaryReader, and BinaryWriter classes 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, I will invite you 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 Datafile Adapter project.

17.9 FailedRecordCreationException.cs

```
1     using System;
2
3     public class FailedRecordCreationException : Exception {
4
5         public FailedRecordCreationException() : base("Failed Record Creation Exception") { }
6
7         public FailedRecordCreationException(String message) : base(message) { }
8
9         public FailedRecordCreationException(String message, Exception inner_exception) :
```


© 2008 Rick Miller — All Rights Reserved

```
10                                                base(message, inner_exception) { }
11    }
```

17.10 InvalidDataFileExcepton.cs

```
1     using System;
2
3     public class InvalidDataFileException : Exception {
4
5         public InvalidDataFileException() : base("Invalid Data File Exception") { }
6
7         public InvalidDataFileException(String message) : base(message) { }
8
9         public InvalidDataFileException(String message, Exception inner_exception) :
10                                             base(message, inner_exception) { }
11    }
```

17.11 NewDatafileException.cs

```
1     using System;
2
3     public class NewDataFileException : Exception {
4
5         public NewDataFileException() : base("New Data File Exception") { }
6
7         public NewDataFileException(String message) : base(message) { }
8
9         public NewDataFileException(String message, Exception inner_exception) :
10                                         base(message, inner_exception) { }
11    }
```

17.12 RecordNotFoundException.cs

```
1     using System;
2
3     public class RecordNotFoundException : Exception {
4
5         public RecordNotFoundException() : base("Record Not Found Exception") { }
6
7         public RecordNotFoundException(String message) : base(message) { }
8
9         public RecordNotFoundException(String message, Exception inner_exception) :
10                                            base(message, inner_exception) { }
11    }
```

17.13 SecurityException.cs

```
1     using System;
2
3     public class SecurityException : Exception {
4
5         public SecurityException() : base("Security Exception") { }
6
7         public SecurityException(String message) : base(message) { }
8
9         public SecurityException(String message, Exception inner_exception) :
10                                      base(message, inner_exception) { }
11    }
```

17.14 LegacyDatafileInterface.cs

```
1     using System;
2
3     public interface LegacyDatafileInterface {
4
5
6       /// <summary>
7       /// Read the record indicated by the rec_no and return a string array
8       /// were each element contains a field value.
9       /// </summary>
10      /// <param name="rec_no"></param>
11      /// <returns>A string array containing the record fields</returns>
12      /// <exception cref="RecordNotFoundException"</exception>
13      String[] ReadRecord(long rec_no);
14
15
16      /// <summary>
17      /// Update a record's fields. The record must be locked with the lockRecord()
18      /// method and the lock_token must be valid. The value for field n appears in
```

 © 2008 Rick Miller — All Rights Reserved

```
19     /// element record[n].
20     /// </summary>
21     /// <param name="rec_no"></param>
22     /// <param name="record"></param>
23     /// <param name="lock_token"></param>
24     /// <exception cref="RecordNotFoundException"></exception>
25     /// <exception cref="SecurityException"></exception>
26     void UpdateRecord(long rec_no, String[] record, long lock_token);
27
28
29     /// <summary>
30     /// Marks a record for deletion by setting the deleted field to 1. The lock_token
31     /// must be valid otherwise a SecurityException is thrown.
32     /// </summary>
33     /// <param name="rec_no"></param>
34     /// <param name="lock_token"></param>
35     /// <exception cref="RecordNotFoundException" ></exception>
36     /// <exception cref="SecurityException"></exception>
37     void DeleteRecord(long rec_no, long lock_token);
38
39     /// <summary>
40     /// Creates a new datafile record and returns the record number.
41     /// </summary>
42     /// <param name="record"></param>
43     /// <returns>The record number of the newly created record</returns>
44     /// <exception cref="FailedRecordCreationException"></exception>
45     long CreateRecord(String[] record);
46
47
48     /// <summary>
49     /// Locks a record for updates and deletes and returns an integer
50     /// representing a lock token.
51     /// </summary>
52     /// <param name="rec_no"></param>
53     /// <returns>Lock token</returns>
54     /// <exception cref="RecordNotFoundException"></exception>
55     long LockRecord(long rec_no);
56
57
58     /// <summary>
59     /// Unlocks a previously locked record. The lock_token must be valid or a
60     /// SecurityException is thrown.
61     /// </summary>
62     /// <param name="rec_no"></param>
63     /// <param name="lock_token"></param>
64     /// <exception cref="SecurityException"></exception>
65     void UnlockRecord(long rec_no, long lock_token);
66
67
68     /// <summary>
69     ///  Searches the records in the datafile for records that match the String
70     /// values of search_criteria. search_criteria[n] contains the search value
71     /// applied against field n.
72     /// </summary>
73     /// <param name="search_criteria"></param>
74     /// <returns>An array of longs containing the matched record numbers</returns>
75     long[] SearchRecords(String[] search_criteria);
76
77   }//end interface definition
```

17.15 DataFileAdapter.cs

```
1    using System;
2    using System.IO;
3    using System.Text;
4    using System.Threading;
5    using System.Collections;
6    using System.Collections.Generic;
7
8
9    public class DataFileAdapter : LegacyDatafileInterface {
10
11        /*************************************
12        * Constants
13        **************************************/
14
15       private const short FILE_IDENTIFIER = 378;
16       private const int HEADER_LENGTH = 54;
17       private const int RECORDS_START = 54;
18       private const int RECORD_LENGTH = 130;
19       private const int FIELD_COUNT = 7;
```

 © 2008 Rick Miller — All Rights Reserved

```
20
21       private const short DELETED_FIELD_LENGTH = 1;
22       private const short TITLE_FIELD_LENGTH = 50;
23       private const short AUTHOR_FIELD_LENGTH = 50;
24       private const short PUB_CODE_FIELD_LENGTH = 4;
25       private const short ISBN_FIELD_LENGTH = 13;
26       private const short PRICE_FIELD_LENGTH = 8;
27       private const short QOH_FIELD_LENGTH = 4;
28
29       private const String DELETED_STRING = "deleted";
30       private const String TITLE_STRING = "title";
31       private const String AUTHOR_STRING = "author";
32       private const String PUB_CODE_STRING = "pub_code";
33       private const String ISBN_STRING = "ISBN";
34       private const String PRICE_STRING = "price";
35       private const String QOH_STRING = "qoh";
36
37       private const int TITLE_FIELD = 0;
38       private const int AUTHOR_FIELD = 1;
39       private const int PUB_CODE_FIELD = 2;
40       private const int ISBN_FIELD = 3;
41       private const int PRICE_FIELD = 4;
42       private const int QOH_FIELD = 5;
43
44       private const int VALID = 0;
45       private const int DELETED = 1;
46
47       /********************************************************
48        * Private Instance Fields
49        ********************************************************/
50       private String _filename = null;
51       private BinaryReader _reader = null;
52       private BinaryWriter _writer = null;
53       private long _record_count = 0;
54       private Hashtable _locked_records_map = null;
55       private Random _token_maker = null;
56       private long _current_record_number = 0;
57       private bool _debug = false;
58
59       /*******************************************************
60        * Properties
61        *******************************************************/
62       public long RecordCount {
63          get { return _record_count; }
64        }
65
66       /********************************************************
67        * Instance Methods
68        ********************************************************/
69
70       /// <summary>
71       /// Constructor
72       /// </summary>
73       /// <param name="filename"></param>
74       /// <exception cref="InvalidDataFileException"></exception>
75       public DataFileAdapter(String filename) {
76         try {
77           _filename = filename;
78           if(File.Exists(_filename)){
79             _reader = new BinaryReader(File.Open(filename, FileMode.Open));
80             if ((_reader.BaseStream.Length >= HEADER_LENGTH) && (_reader.ReadInt16() == FILE_IDENTIFIER)) {
81                  // it's a valid data file
82                Console.WriteLine(_filename + " is a valid data file...");
83                _record_count = ((_reader.BaseStream.Length - HEADER_LENGTH) / RECORD_LENGTH);
84                Console.WriteLine("Record count is: " + _record_count);
85                InitializeVariables();
86                _reader.Close();
87               } else if (_reader.BaseStream.Length == 0) { // The file's empty - make it a data file
88                       _reader.Close();
89                       WriteHeader(FileMode.Open);
90                       InitializeVariables();
91                      } else {
92                         _reader.BaseStream.Seek(0, SeekOrigin.Begin);
93                         if (_reader.ReadInt16() != FILE_IDENTIFIER) {
94                           _reader.Close();
95                           Console.WriteLine("Invalid data file. Closing file.");
96                           throw new InvalidDataFileException("Invalid data file identifier...");
97                         }
98                      }
99           }else {
100            CreateNewDataFile(_filename);
```

 © 2008 Rick Miller — All Rights Reserved

```
101          }
102        }catch (ArgumentException e) {
103            if(_debug){ Console.WriteLine(e.ToString()); }
104            throw new InvalidDataFileException("Invalid argument.",e);
105          }
106         catch (EndOfStreamException e) {
107            if(_debug){ Console.WriteLine(e.ToString()); }
108            throw new InvalidDataFileException("End of stream exception.",e);
109          }
110         catch (ObjectDisposedException e) {
111            if(_debug){ Console.WriteLine(e.ToString()); }
112            throw new InvalidDataFileException("BinaryReader not initialized.",e);
113          }
114         catch (IOException e) {
115            if(_debug){ Console.WriteLine(e.ToString()); }
116            throw new InvalidDataFileException("General IOException",e);
117          }
118         catch (Exception e) {
119            if(_debug){ Console.WriteLine(e.ToString()); }
120            throw new InvalidDataFileException("General Exception",e);
121          }
122          finally {
123            if (_reader != null) {
124              _reader.Close();
125            }
126          }
127      } // end constructor
128
129
130      /// <summary>
131      /// Default Constructor
132      /// </summary>
133      /// <exception cref="InvalidDataFileException"></exception>
134      public DataFileAdapter():this("books.dat"){ }
135
136
137      /// <summary>
138      /// Create new file
139      /// </summary>
140      /// <param name="filename"></param>
141      /// <exception cref="NewDataFileException"></exception>
142      public void CreateNewDataFile(String filename) {
143        try {
144          _filename = filename;
145          WriteHeader(FileMode.Create);
146          InitializeVariables();
147        } catch (Exception e) {
148            if(_debug) { Console.WriteLine(e); }
149            throw new NewDataFileException(e.ToString());
150          }
151      } // end createNewDataFile method
152
153
154      /// <summary>
155      /// Read the record indicated by the rec_no and return a string array
156      /// were each element contains a field value.
157      /// </summary>
158      /// <param name="rec_no"></param>
159      /// <returns>A populated string array containing record field values</returns>
160      /// <exception cref="RecordNotFoundException"></exception>
161      public String[] ReadRecord(long rec_no) {
162        String[] temp_string = null;
163        if ((rec_no < 0) || (rec_no > _record_count)) {
164          if(_debug){ Console.WriteLine("From ReadRecord(): Requested record out of range!"); }
165           throw new RecordNotFoundException("From ReadRecord(): Requested record out of range");
166        } else {
167            try {
168              _reader = new BinaryReader(File.Open(_filename, FileMode.Open));
169              GotoRecordNumber(_reader, rec_no);
170              if (_reader.ReadByte() == DELETED) {
171                if(_debug){ Console.WriteLine("From ReadRecord(): Record number " + rec_no +
172                                                        " has been deleted!"); }
173                throw new RecordNotFoundException("Record " + rec_no + " deleted!");
174              } else {
175                  temp_string = RecordBytesToStringArray(_reader, rec_no);
176                }
177            } catch (ArgumentException e) {
178            if(_debug){ Console.WriteLine(e.ToString()); }
179            throw new RecordNotFoundException("Invalid argument.",e);
180          }
181         catch (EndOfStreamException e) {
```


© 2008 Rick Miller — All Rights Reserved

```
182            if(_debug){ Console.WriteLine(e.ToString()); }
183            throw new RecordNotFoundException("End of stream exception.",e);
184          }
185         catch (ObjectDisposedException e) {
186            if(_debug){ Console.WriteLine(e.ToString()); }
187            throw new RecordNotFoundException("BinaryReader not initialized.",e);
188          }
189         catch (IOException e) {
190            if(_debug){ Console.WriteLine(e.ToString()); }
191            throw new RecordNotFoundException("General IOException",e);
192          }
193         catch (Exception e) {
194            if(_debug){ Console.WriteLine(e.ToString()); }
195            throw new RecordNotFoundException("General Exception",e);
196          }
197              finally {
198                  if (_reader != null) {
199                      _reader.Close();
200                  }
201              }
202          } // end else
203        return temp_string;
204      } // end readRecord()
205
206
207      /// <summary>
208      /// Update a record's fields. The record must be locked with the lockRecord()
209      /// method and the lock_token must be valid. The value for field n appears in
210      /// element record[n]. The call to updateRecord() MUST be preceeded by a call
211      /// to lockRecord() and followed by a call to unlockRecord()
212      /// </summary>
213      /// <param name="rec_no"></param>
214      /// <param name="record"></param>
215      /// <param name="lock_token"></param>
216      /// <exception cref="RecordNotFoundException"></exception>
217      /// <exception cref="SecurityException"></exception>
218      public void UpdateRecord(long rec_no, String[] record, long lock_token) {
219        if (lock_token != ((long)_locked_records_map[rec_no])) {
220          if(_debug){ Console.WriteLine("From UpdateRecord(): Invalid  update record lock token."); }
221          throw new SecurityException("From UpdateRecord(): Invalid update record lock token.");
222        } else {
223            try {
224              _writer = new BinaryWriter(File.Open(_filename, FileMode.Open));
225              GotoRecordNumber(_writer, rec_no); //i.e., goto indicated record
226              _writer.Write((byte)0);
227              _writer.Write(StringToPaddedByteField(record[TITLE_FIELD], TITLE_FIELD_LENGTH));
228              _writer.Write(StringToPaddedByteField(record[AUTHOR_FIELD], AUTHOR_FIELD_LENGTH));
229              _writer.Write(Int16.Parse(record[PUB_CODE_FIELD]));
230              _writer.Write(StringToPaddedByteField(record[ISBN_FIELD], ISBN_FIELD_LENGTH));
231              _writer.Write(StringToPaddedByteField(record[PRICE_FIELD], PRICE_FIELD_LENGTH));
232              _writer.Write(Int16.Parse(record[QOH_FIELD]));
233              _current_record_number = rec_no;
234            } catch (ArgumentException e) {
235               if(_debug){ Console.WriteLine(e.ToString()); }
236               throw new RecordNotFoundException("Invalid argument.",e);
237              }
238             catch (EndOfStreamException e) {
239              if(_debug){ Console.WriteLine(e.ToString()); }
240              throw new RecordNotFoundException("End of stream exception.",e);
241             }
242             catch (ObjectDisposedException e) {
243              if(_debug){ Console.WriteLine(e.ToString()); }
244              throw new RecordNotFoundException("BinaryReader not initialized.",e);
245              }
246             catch (IOException e) {
247              if(_debug){ Console.WriteLine(e.ToString()); }
248              throw new RecordNotFoundException("General IOException",e);
249             }
250             catch (Exception e) {
251              if(_debug){ Console.WriteLine(e.ToString()); }
252              throw new RecordNotFoundException("General Exception",e);
253             }
254              finally {
255                if (_writer != null) {
256                  _writer.Close();
257                }
258              }
259          }// end else
260      }// end updateRecord()
261
262
```

 © 2008 Rick Miller — All Rights Reserved

```
263       /// <summary>
264       /// Marks a record for deletion by setting the deleted field to 1. The lock_token
265       /// must be valid otherwise a SecurityException is thrown.
266       /// </summary>
267       /// <param name="rec_no"></param>
268       /// <param name="lock_token"></param>
269       /// <exception cref="RecordNotFoundException"></exception>
270       /// <exception cref="SecurityException"></exception>
271       public void DeleteRecord(long rec_no, long lock_token) {
272         if (lock_token != (long)_locked_records_map[rec_no]) {
273           Console.WriteLine("From DeleteRecord(): Invalid delete record lock token.");
274           throw new SecurityException("From DeleteRecord(): Invalid delete record lock token.");
275         } else {
276             try {
277               _writer = new BinaryWriter(File.Open(_filename, FileMode.Open));
278               GotoRecordNumber(_writer, rec_no); // goto record indicated
279               _writer.Write((byte)1);  // mark for deletion
280             } catch (ArgumentException e) {
281               if(_debug){ Console.WriteLine(e.ToString()); }
282               throw new RecordNotFoundException("Invalid argument.",e);
283              }
284              catch (EndOfStreamException e) {
285               if(_debug){ Console.WriteLine(e.ToString()); }
286               throw new RecordNotFoundException("End of stream exception.",e);
287              }
288              catch (ObjectDisposedException e) {
289               if(_debug){ Console.WriteLine(e.ToString()); }
290               throw new RecordNotFoundException("BinaryReader not initialized.",e);
291              }
292              catch (IOException e) {
293               if(_debug){ Console.WriteLine(e.ToString()); }
294               throw new RecordNotFoundException("General IOException",e);
295              }
296             catch (Exception e) {
297               if(_debug){ Console.WriteLine(e.ToString()); }
298               throw new RecordNotFoundException("General Exception",e);
299             }
300               finally {
301                 if (_writer != null) {
302                    _writer.Close();
303                 }
304               }
305           }// end else
306       }// end deleteRecord()
307
308
309       /// <summary>
310       /// Creates a new datafile record and returns the record number.
311       /// </summary>
312       /// <param name="record"></param>
313       /// <returns> The record number of the newly created record</returns>
314       /// <exception cref="FailedRecordCreationException"></exception>
315       public long CreateRecord(String[] record) {
316         try {
317           _writer = new BinaryWriter(File.Open(_filename, FileMode.Open));
318           GotoRecordNumber(_writer, _record_count); //i.e., goto end of file
319           _writer.Write((byte)0);
320           _writer.Write(StringToPaddedByteField(record[TITLE_FIELD], TITLE_FIELD_LENGTH));
321           _writer.Write(StringToPaddedByteField(record[AUTHOR_FIELD], AUTHOR_FIELD_LENGTH));
322           _writer.Write(Int16.Parse(record[PUB_CODE_FIELD]));
323           _writer.Write(StringToPaddedByteField(record[ISBN_FIELD], ISBN_FIELD_LENGTH));
324           _writer.Write(StringToPaddedByteField(record[PRICE_FIELD], PRICE_FIELD_LENGTH));
325           _writer.Write(Int16.Parse(record[QOH_FIELD]));
326           _current_record_number = ++_record_count;
327         } catch (ArgumentException e) {
328             if(_debug){ Console.WriteLine(e.ToString()); }
329             throw new FailedRecordCreationException("Invalid argument.",e);
330           }
331          catch (EndOfStreamException e) {
332             if(_debug){ Console.WriteLine(e.ToString()); }
333             throw new FailedRecordCreationException("End of stream exception.",e);
334           }
335          catch (ObjectDisposedException e) {
336             if(_debug){ Console.WriteLine(e.ToString()); }
337             throw new FailedRecordCreationException("BinaryReader not initialized.",e);
338           }
339          catch (IOException e) {
340             if(_debug){ Console.WriteLine(e.ToString()); }
341             throw new FailedRecordCreationException("General IOException",e);
342           }
343          catch (Exception e) {
```

 © 2008 Rick Miller — All Rights Reserved

```
344           if(_debug){ Console.WriteLine(e.ToString()); }
345           throw new FailedRecordCreationException("General Exception",e);
346         }
347         finally {
348           if (_writer != null) {
349             _writer.Close();
350           }
351         }
352       return _current_record_number;
353     } // end CreateRecord()
354
355
356     /// <summary>
357     /// Locks a record for updates and deletes  - returns an integer
358     /// representing a lock token.
359     /// </summary>
360     /// <param name="rec_no"></param>
361     /// <returns></returns>
362     /// <exception cref="RecordNotFoundException"></exception>
363     public long LockRecord(long rec_no) {
364       long lock_token = 0;
365       if ((rec_no < 0) || (rec_no > _record_count)) {
366         if(_debug){ Console.WriteLine("Record cannot be locked. Not in valid range."); }
367         throw new RecordNotFoundException("Record cannot be locked. Not in valid range.");
368       } else {
369           lock (_locked_records_map) {
370           while (_locked_records_map.ContainsKey(rec_no)) {
371             try {
372               Monitor.Wait(_locked_records_map);
373             } catch (Exception) { }
374           }
375           lock_token = (long)_token_maker.Next();
376           _locked_records_map.Add(rec_no, lock_token);
377           } // end lock
378         } // end else
379       return lock_token;
380     } // end LockRecord()
381
382
383     /// <summary>
384     /// Unlocks a previously locked record. The lock_token must be valid or a
385     /// SecurityException is thrown.
386     /// </summary>
387     /// <param name="rec_no"></param>
388     /// <param name="lock_token"></param>
389     /// <exception cref="SecurityException"></exception>
390     public void UnlockRecord(long rec_no, long lock_token) {
391       lock (_locked_records_map) {
392         if (_locked_records_map.Contains(rec_no)) {
393           if (lock_token == ((long)_locked_records_map[rec_no])) {
394              _locked_records_map.Remove(rec_no);
395              Monitor.Pulse(_locked_records_map);
396           } else {
397               if(_debug){ Console.WriteLine("From UnlockRecord(): Invalid lock token."); }
398               throw new SecurityException("From UnlockRecord(): Invalid lock token");
399             }
400           } else {
401               if(_debug){ Console.WriteLine("From UnlockRecord(): Invalid record number."); }
402               throw new SecurityException("From UnlockRecord(): Invalid record number.");
403             }
404         }
405     }// end UnlockRecord()
406
407
408     /// <summary>
409     /// Searches the records in the datafile for records that match the String
410     /// values of search_criteria. search_criteria[n] contains the search value
411     /// applied against field n. Data files can be searched for Title & Author.
412     /// </summary>
413     /// <param name="search_criteria"></param>
414     /// <returns>An array of long values each indicating a record number match</returns>
415     public long[] SearchRecords(String[] search_criteria) {
416       List<long> hit_list = new List<long>();
417       for (long i = 0; i < _record_count; i++) {
418         try {
419           if (ThereIsAMatch(search_criteria, ReadRecord(i))) {
420              hit_list.Add(i);
421           }
422         } catch (RecordNotFoundException) { } // ignore deleted records
423       } // end for
424       long[] hits = new long[hit_list.Count];
```

 © 2008 Rick Miller — All Rights Reserved

```
425         for (int i = 0; i < hits.Length; i++) {
426           hits[i] = hit_list[i];
427         }
428         return hits;
429       } // end SearchRecords()
430
431
432       /// <summary>
433       /// ThereIsAMatch() is a utility method that actually performs
434       /// the record search. Implements an implied OR/AND search by detecting
435       /// the first character of the Title criteria element.
436       /// </summary>
437       /// <param name="search_criteria"></param>
438       /// <param name="record"></param>
439       /// <returns>A boolean value indicating true if there is a match or false otherwise.</returns>
440       private bool ThereIsAMatch(String[] search_criteria, String[] record) {
441         bool match_result = false;
442         int TITLE = 0;
443         int AUTHOR = 1;
444         for (int i = 0; i < search_criteria.Length; i++) {
445           if ((search_criteria[i].Length == 0) || (record[i + 1].StartsWith(search_criteria[i]))) {
446              match_result = true;
447              break;
448           } //end if
449         } //end for
450
451         if (((search_criteria[TITLE].Length > 1) && (search_criteria[AUTHOR].Length >= 1)) &&
452                                                             (search_criteria[TITLE][0] == '&')) {
453             if (record[TITLE + 1].StartsWith(search_criteria[TITLE].Substring(1,
454
search_criteria[TITLE].Length).Trim())
455                  && record[AUTHOR + 1].StartsWith(search_criteria[AUTHOR])) {
456                  match_result = true;
457            } else {
458                 match_result = false;
459              }
460         } // end  outer if
461         return match_result;
462       } // end thereIsAMatch()
463
464
465       /// <summary>
466       /// GotoRecordNumber - utility function that handles the messy
467       /// details of seeking a particular record.
468       /// </summary>
469       /// <param name="record_number"></param>
470       /// <exception cref="RecordNotFoundException"></exception>
471       private void GotoRecordNumber(BinaryReader reader, long record_number) {
472         if ((record_number < 0) || (record_number > _record_count)) {
473           throw new RecordNotFoundException();
474         } else {
475             try {
476               reader.BaseStream.Seek(RECORDS_START + (record_number * RECORD_LENGTH), SeekOrigin.Begin);
477             } catch (EndOfStreamException e) {
478                 if(_debug){ Console.WriteLine(e.ToString()); }
479                 throw new RecordNotFoundException("End of stream exception.",e);
480               }
481               catch (ObjectDisposedException e) {
482                 if(_debug){ Console.WriteLine(e.ToString()); }
483                 throw new RecordNotFoundException("BinaryReader not initialized.",e);
484               }
485               catch (IOException e) {
486                 if(_debug){ Console.WriteLine(e.ToString()); }
487                 throw new RecordNotFoundException("General IOException",e);
488               }
489               catch (Exception e) {
490                 if(_debug){ Console.WriteLine(e.ToString()); }
491                 throw new RecordNotFoundException("General Exception",e);
492               }
493           }// end else
494       } // end GotoRecordNumber()
495
496
497       /// <summary>
498       /// GotoRecordNumber - overloaded utility function that handles the messy
499       /// details of seeking a particular record.
500       /// </summary>
501       /// <param name="record_number"></param>
502       /// <exception cref="RecordNotFoundException"></exception>
503       private void GotoRecordNumber(BinaryWriter writer, long record_number) {
504         if ((record_number < 0) || (record_number > _record_count)) {
```


© 2008 Rick Miller — All Rights Reserved

```
505          throw new RecordNotFoundException();
506        } else {
507           try {
508             writer.BaseStream.Seek(RECORDS_START + (record_number * RECORD_LENGTH), SeekOrigin.Begin);
509           } catch (EndOfStreamException e) {
510               if(_debug){ Console.WriteLine(e.ToString()); }
511               throw new RecordNotFoundException("End of stream exception.",e);
512             }
513             catch (ObjectDisposedException e) {
514               if(_debug){ Console.WriteLine(e.ToString()); }
515               throw new RecordNotFoundException("BinaryReader not initialized.",e);
516             }
517             catch (IOException e) {
518               if(_debug){ Console.WriteLine(e.ToString()); }
519               throw new RecordNotFoundException("General IOException",e);
520             }
521             catch (Exception e) {
522               if(_debug){ Console.WriteLine(e.ToString()); }
523               throw new RecordNotFoundException("General Exception",e);
524             }
525         }// end else
526       } // end GotoRecordNumber()
527
528
529       /// <summary>
530       /// stringToPaddedByteField - pads the field to maintain fixed
531       /// field length.
532       /// </summary>
533       /// <param name="s"></param>
534       /// <param name="field_length"></param>
535       /// <returns>A populated byte array containing the string value padded with spaces</returns>
536       protected byte[] StringToPaddedByteField(String s, int field_length) {
537         byte[] byte_field = new byte[field_length];
538         if (s.Length <= field_length) {
539            for (int i = 0; i < s.Length; i++) {
540              byte_field[i] = (byte)s[i];
541            }
542            for (int i = s.Length; i < field_length; i++) {
543              byte_field[i] = (byte)' '; //pad the field
544            }
545         } else {
546             for (int i = 0; i < field_length; i++) {
547               byte_field[i] = (byte)s[i];
548             }
549           }
550         return byte_field;
551       } // end StringToPaddedByteField()
552
553
554       /// <summary>
555       /// RecordBytesToStringArray - reads an array of bytes from a data file
556       /// and converts them to an array of Strings. The first element of the
557       /// returned array is the record number. The length of the byte array
558       /// argument is RECORD_LENGTH -1.
559       /// </summary>
560       /// <param name="record_number"></param>
561       /// <returns></returns>
562       private String[] RecordBytesToStringArray(BinaryReader reader, long record_number) {
563         String[] string_array = new String[FIELD_COUNT];
564         char[] title = new char[TITLE_FIELD_LENGTH];
565         char[] author = new char[AUTHOR_FIELD_LENGTH];
566         char[] isbn = new char[ISBN_FIELD_LENGTH];
567         char[] price = new char[PRICE_FIELD_LENGTH];
568         try {
569           string_array[0] = record_number.ToString();
570           reader.Read(title, 0, title.Length);
571           string_array[TITLE_FIELD + 1] = new String(title).Trim();
572           reader.Read(author, 0, author.Length);
573           string_array[AUTHOR_FIELD + 1] = new String(author).Trim();
574           string_array[PUB_CODE_FIELD + 1] = (reader.ReadInt16()).ToString();
575           reader.Read(isbn, 0, isbn.Length);
576           string_array[ISBN_FIELD + 1] = new String(isbn);
577           reader.Read(price, 0, price.Length);
578           string_array[PRICE_FIELD + 1] = new String(price).Trim();
579           string_array[QOH_FIELD + 1] = (reader.ReadInt16()).ToString();
580         } catch (IOException e) {
581             Console.WriteLine(e.ToString());
582           }
583         return string_array;
584       } // end recordBytesToStringArray()
585
```

 © 2008 Rick Miller — All Rights Reserved

```
586
587      /// <summary>
588      /// Writes the header information into a data file
589      /// </summary>
590      /// <exception cref="InvalidDataFileException"></exception>
591      private void WriteHeader(FileMode file_mode) {
592        try {
593          if (_writer != null) {
594            _writer.Close();
595          }
596          _writer = new BinaryWriter(File.Open(_filename, file_mode));
597          _writer.Seek(0, SeekOrigin.Begin);
598          _writer.Write(FILE_IDENTIFIER);
599          _writer.Write(DELETED_STRING.ToCharArray());
600          _writer.Write(DELETED_FIELD_LENGTH);
601          _writer.Write(TITLE_STRING.ToCharArray());
602          _writer.Write(TITLE_FIELD_LENGTH);
603          _writer.Write(AUTHOR_STRING.ToCharArray());
604          _writer.Write(AUTHOR_FIELD_LENGTH);
605          _writer.Write(PUB_CODE_STRING.ToCharArray());
606          _writer.Write(PUB_CODE_FIELD_LENGTH);
607          _writer.Write(ISBN_STRING.ToCharArray());
608          _writer.Write(ISBN_FIELD_LENGTH);
609          _writer.Write(PRICE_STRING.ToCharArray());
610          _writer.Write(PRICE_FIELD_LENGTH);
611          _writer.Write(QOH_STRING.ToCharArray());
612          _writer.Write(QOH_FIELD_LENGTH);
613          _writer.Flush();
614        }catch (ArgumentException e) {
615            if(_debug){ Console.WriteLine(e.ToString()); }
616            throw new InvalidDataFileException("Invalid argument.",e);
617          }
618         catch (EndOfStreamException e) {
619            if(_debug){ Console.WriteLine(e.ToString()); }
620            throw new InvalidDataFileException("End of stream exception.",e);
621          }
622         catch (ObjectDisposedException e) {
623            if(_debug){ Console.WriteLine(e.ToString()); }
624            throw new InvalidDataFileException("BinaryReader not initialized.",e);
625          }
626         catch (IOException e) {
627            if(_debug){ Console.WriteLine(e.ToString()); }
628            throw new InvalidDataFileException("General IOException",e);
629          }
630         catch (Exception e) {
631            if(_debug){ Console.WriteLine(e.ToString()); }
632            throw new InvalidDataFileException("General Exception",e);
633          }
634          finally {
635            if (_writer != null) {
636              _writer.Close();
637            }
638          }
639      } // end WriteHeader()
640
641
642      /// <summary>
643      /// readHeader - reads the header bytes and converts them to
644      /// a string
645      /// </summary>
646      /// <returns> A String containing the file header information</returns>
647      /// <exception cref="InvalidDataFileException"></exception>
648      public String ReadHeader() {
649        StringBuilder sb = new StringBuilder();
650        char[] deleted = new char[DELETED_STRING.Length];
651        char[] title = new char[TITLE_STRING.Length];
652        char[] author = new char[AUTHOR_STRING.Length];
653        char[] pub_code = new char[PUB_CODE_STRING.Length];
654        char[] isbn = new char[ISBN_STRING.Length];
655        char[] price = new char[PRICE_STRING.Length];
656        char[] qoh = new char[QOH_STRING.Length];
657        try {
658          _reader = new BinaryReader(File.Open(_filename, FileMode.Open));
659          _reader.BaseStream.Seek(0, SeekOrigin.Begin);
660          sb.Append(_reader.ReadInt16() + " ");
661          _reader.Read(deleted, 0, deleted.Length);
662          sb.Append(new String(deleted) + " ");
663          sb.Append(_reader.ReadInt16() + " ");
664          _reader.Read(title, 0, title.Length);
665          sb.Append(new String(title) + " ");
666          sb.Append((_reader.ReadInt16()) + " ");
```

 © 2008 Rick Miller — All Rights Reserved

```
667         _reader.Read(author, 0, author.Length);
668         sb.Append(new String(author) + " ");
669         sb.Append((_reader.ReadInt16()) + " ");
670         _reader.Read(pub_code, 0, pub_code.Length);
671         sb.Append(new String(pub_code) + " ");
672         sb.Append((_reader.ReadInt16()) + " ");
673         _reader.Read(isbn, 0, isbn.Length);
674         sb.Append(new String(isbn) + " ");
675         sb.Append((_reader.ReadInt16()) + " ");
676         _reader.Read(price, 0, price.Length);
677         sb.Append(new String(price) + " ");
678         sb.Append((_reader.ReadInt16()) + " ");
679         _reader.Read(qoh, 0, qoh.Length);
680         sb.Append(new String(qoh) + " ");
681         sb.Append((_reader.ReadInt16()) + " ");
682       } catch (ArgumentException e) {
683           if(_debug){ Console.WriteLine(e.ToString()); }
684           throw new InvalidDataFileException("Invalid argument.",e);
685         }
686        catch (EndOfStreamException e) {
687           if(_debug){ Console.WriteLine(e.ToString()); }
688           throw new InvalidDataFileException("End of stream exception.",e);
689         }
690        catch (ObjectDisposedException e) {
691           if(_debug){ Console.WriteLine(e.ToString()); }
692           throw new InvalidDataFileException("BinaryReader not initialized.",e);
693         }
694        catch (IOException e) {
695           if(_debug){ Console.WriteLine(e.ToString()); }
696           throw new InvalidDataFileException("General IOException",e);
697         }
698        catch (Exception e) {
699           if(_debug){ Console.WriteLine(e.ToString()); }
700           throw new InvalidDataFileException("General Exception",e);
701         }
702         finally {
703           if (_reader != null) {
704             _reader.Close();
705           }
706         }
707       return sb.ToString();
708     } // end ReadHeader()
709
710
711     /// <summary>
712     /// Utility method used to initialize several important instance fields
713     /// </summary>
714     private void InitializeVariables() {
715       _current_record_number = 0;
716       _locked_records_map = new Hashtable();
717       _token_maker = new Random();
718     }
719
720 } // end DataFileAdapter class definition
```

17.16 AdapterTestApp.cs

```
1    using System;
2
3    public class AdapterTesterApp {
4       public static void Main(){
5         try{
6           DataFileAdapter adapter = new DataFileAdapter("books.dat");
7           String[] rec_1 = {"C++ For Artists", "Rick Miller", "0001", "1-932504-02-8", "$59.95", "80"};
8           String[] rec_2 = {"Java For Artists", "Rick Miller", "0002", "1-932504-04-X", "$69.95", "100"};
9           String[] rec_3 = { "C# For Artists", "Rick Miller", "0003", "1-932504-07-9", "$76.00", "567" };
10          String[] rec_4 = { "White Saturn", "Rick Miller", "0004", "1-932504-08-7", "$45.00", "234" };
11
12          String[] search_string = {"Java", " "};
13
14          String[] temp_string = null;
15
16          adapter.CreateRecord(rec_1);
17          adapter.CreateRecord(rec_2);
18          adapter.CreateRecord(rec_3);
19          adapter.CreateRecord(rec_1);
20          adapter.CreateRecord(rec_2);
21          adapter.CreateRecord(rec_3);
22          adapter.CreateRecord(rec_1);
```

 © 2008 Rick Miller — All Rights Reserved

```
23          adapter.CreateRecord(rec_2);
24          adapter.CreateRecord(rec_3);
25
26
27          long lock_token = adapter.LockRecord(2);
28
29          adapter.UpdateRecord(2, rec_2, lock_token);
30          adapter.UnlockRecord(2, lock_token);
31
32          lock_token = adapter.LockRecord(1);
33          adapter.DeleteRecord(1, lock_token);
34          adapter.UnlockRecord(1, lock_token);
35
36          lock_token = adapter.LockRecord(4);
37          adapter.UpdateRecord(4, rec_4, lock_token);
38          adapter.UnlockRecord(4, lock_token);
39
40          long[] search_hits = adapter.SearchRecords(search_string);
41
42          Console.WriteLine(adapter.ReadHeader());
43
44          for(int i=0; i<search_hits.Length; i++){
45            try{
46            temp_string = adapter.ReadRecord(search_hits[i]);
47            for(int j = 0; j<temp_string.Length; j++){
48             Console.Write(temp_string[j] + " ");
49             }
50            Console.WriteLine();
51               }catch(RecordNotFoundException){ }
52          }
53
54          Console.WriteLine("-----------------------------------------");
55          for (int i = 0; i < adapter.RecordCount; i++) {
56            try {
57              temp_string = adapter.ReadRecord(i);
58              for (int j = 0; j < temp_string.Length; j++) {
59                Console.Write(temp_string[j] + " ");
60              }
61              Console.WriteLine();
62            }
63            catch (RecordNotFoundException) {  }
64          }
65        }
66        catch (Exception e) { Console.WriteLine(e.ToString()); }
67      } // end Main()
68   } // end class definition
```

Figure 17-11 shows the results of running the AdapterTestApp program one time. Running it several times back-to-back results in additional records being inserted into the book.dat data file.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\RandomAccess>adaptertestapp
378 deleted 1 title 50 author 50 pub_code 4 ISBN 13 price 8 qoh 4
2 Java For Artists Rick Miller 2 1-932504-04-X $69.95 100
7 Java For Artists Rick Miller 2 1-932504-04-X $69.95 100
-----------------------------------------
0 C++ For Artists Rick Miller 1 1-932504-02-8 $59.95 80
2 Java For Artists Rick Miller 2 1-932504-04-X $69.95 100
3 C++ For Artists Rick Miller 1 1-932504-02-8 $59.95 80
4 White Saturn Rick Miller 4 1-932504-08-7 $45.00 234
5 C# For Artists Rick Miller 3 1-932504-07-9 $76.00 567
6 C++ For Artists Rick Miller 1 1-932504-02-8 $59.95 80
7 Java For Artists Rick Miller 2 1-932504-04-X $69.95 100
8 C# For Artists Rick Miller 3 1-932504-07-9 $76.00 567

C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\RandomAccess>_
```

Figure 17-12: Results of Running Example 17.16 Once

Quick Review

You can conduct random access file I/O with the BinaryReader, BinaryWriter, and FileStream classes. The FileStream class provides a Seek() method that allows you to position the file pointer at any point within a file. As you learned in the previous section, the BinaryReader and BinaryWriter classes provide methods for reading and writing binary, string, byte, and character array data.

© 2008 Rick Miller — All Rights Reserved

Working With Log Files

The System.IO.Log namespace contains classes, structures, interfaces, and enumerations designed to help you create robust event logging services for your programs. Some of the functionality provided by the contents of the System.IO.Log namespace is only available on Microsoft Windows 2003r2 and Windows Vista or later operating systems. These operating systems come with the Common Log File System (CLFS).

The following three examples together implement a simple logging system. It consists of three classes. The first, LogEntry, given in Example 17.17, represents the type of data that will be saved in the log file. The second, Logger, given in Example 17.18, implements the logging functionality with the help of several classes in the System.IO.Log namespace. The third class, LoggerTestApp, given in Example 17.19, tests the Logger class by writing several entries to the log and then reading the log and writing its contents to the console.

17.17 LogEntry.cs

```
1    using System;
2
3    [Serializable]
4    public class LogEntry {
5      private string _subsystem;
6      private int _severity;
7      private string _text;
8      private DateTime _timestamp;
9
10     public DateTime TimeStamp {
11       get { return _timestamp; }
12       set { _timestamp = value; }
13     }
14
15     public string SubSystem {
16       get { return _subsystem; }
17       set { _subsystem = value; }
18     }
19
20     public int Severity {
21       get { return _severity; }
22       set { _severity = value; }
23     }
24
25     public string Text {
26       get { return _text; }
27       set { _text = value; }
28     }
29
30     public LogEntry(DateTime timestamp, string subsystem, int severity, string text){
31       TimeStamp = timestamp;
32       SubSystem = subsystem;
33       Severity = severity;
34       Text = text;
35     }
36
37     public override String ToString(){
38       return TimeStamp.ToString() + " " + SubSystem + " " + Severity + " " + Text;
39     }
40   } // end LogEntry class definition
```

Referring to Example 17-17 — the LogEntry class represents the data that will be captured and written to the log. A log entry will contain a TimeStamp property indicating when the event occurred, a SubSystem property indicating the subsystem of origin, Severity property indicating the severity of the event, and a Text property that contains the string with a detailed description of the event.

17.18 Logger.cs

```
1    using System;
2    using System.IO;
3    using System.IO.Log;
4    using System.Collections.Generic;
5    using System.Text;
6    using System.Runtime.Serialization.Formatters.Binary;
7
8    public class Logger {
9      private string _logfilename;
10     private FileRecordSequence _sequence;
11     private SequenceNumber _previous;
12
13
14     public Logger(string logfilename){
```


© 2008 Rick Miller — All Rights Reserved

```
15        _logfilename = logfilename;
16        _sequence = new FileRecordSequence(logfilename, FileAccess.ReadWrite);
17        _previous = SequenceNumber.Invalid;
18      }
19
20      public Logger():this("logfile.log"){ }
21
22      public void Append(LogEntry entry){
23        _previous = _sequence.Append(ToArraySegment(entry), SequenceNumber.Invalid,
24                                     _previous, RecordAppendOptions.ForceFlush);
25      }
26
27      public ArraySegment<byte> ToArraySegment(LogEntry entry) {
28        MemoryStream stream = new MemoryStream();
29        BinaryFormatter formatter = new BinaryFormatter();
30        formatter.Serialize(stream, entry);
31        stream.Flush();
32        return new ArraySegment<byte>(stream.GetBuffer());
33      }
34
35      public String GetLogRecords() {
36        StringBuilder sb = new StringBuilder();
37        BinaryFormatter formatter = new BinaryFormatter();
38        IEnumerable<LogRecord> records = _sequence.ReadLogRecords(_sequence.BaseSequenceNumber,
39                                                               LogRecordEnumeratorType.Next);
40        foreach (LogRecord record in records) {
41          LogEntry entry = (LogEntry) formatter.Deserialize(record.Data);
42          sb.Append(entry.ToString() + "\r\n");
43        }
44        return sb.ToString();
45      }
46
47      public void Dispose(){
48        _sequence.Dispose();
49      }
50    } // end class definition
```

Referring to the Example 17.18 — note that the Logger class uses a host of classes found in other namespaces. From the System.IO.Log namespace it uses the FileRecordSequence and SequenceNumber classes. The FileRecordSequece represents a sequence of log records stored in a simple file. SequenceNumbers are not numbers per se. They represent unique pointers from one log entry to the next within a sequence of log entries.

The Logger.Append() method on line 22 takes a LogEntry reference and in turn calls the FileRecordSequence.Append() method, which actually does the heavy lifting. The FileRecordSequence.Append() method has several overloaded variations. The one I use here requires that the log data being written be presented to it in the first argument as an array segment of bytes. *(i.e.*, ArraySegment<byte>) You'll find the ArraySegment generic structure in the System namespace. The Logger.ToArraySegment() method beginning on line 27 does the dirty work of converting a LogEntry object to a ArraySegment<byte> object.

The Logger.GetLogRecords() method on line 35 uses the FileRecordSequence.ReadLogRecords() method to read the records, converts them back into LogEntry objects, appends their string representation to a StringBuilder object, and ultimately returns the whole lot of them as one long string.

17.19 LoggerTestApp.cs

```
1     using System;
2     using System.Collections.Generic;
3     using System.Text;
4
5      public class LoggeTestApp {
6        static void Main(string[] args) {
7          Logger logger = new Logger();
8          LogEntry entry1 = new LogEntry(DateTime.Now, "Reactor Coolant", 3, "Main coolant pump speed limited");
9          LogEntry entry2 = new LogEntry(DateTime.Now, "Main Engine", 3, "Main condenser loss of vacuum");
10         LogEntry entry3 = new LogEntry(DateTime.Now, "Reactor Coolant", 3, "Main coolant pump speed limited");
11         LogEntry entry4 = new LogEntry(DateTime.Now, "Reactor", 1, "Loss of control rod control");
12         LogEntry entry5 = new LogEntry(DateTime.Now, "Reactor Coolant", 3, "Main coolant pump speed limited");
13
14         logger.Append(entry1);
15         logger.Append(entry2);
16         logger.Append(entry3);
17         logger.Append(entry4);
18         logger.Append(entry5);
19         Console.Write(logger.GetLogRecords());
20         logger.Dispose();
21       } // end Main()
22    } // end class definition
```


© 2008 Rick Miller — All Rights Reserved

Referring to Example 17.19 — the LoggerTestApp creates five LogEntry objects and calls the Logger.Append() method to insert each entry into the log. It then calls the Logger.GetLogRecords() and prints the results to the console.

To compile this program on Windows XP you'll need to do a couple of things. First, you'll need to have installed the .NET Framework 3.0 Redistributable. Second, locate the System.IO.Log.dll in the C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0 directory and add this path to your path environment variable. (See Creating Environment Variables in Chapter 2.) Once you set your path you'll need to compile the source files with the `/reference` switch to compile the files along with the System.IO.Log.dll like so:

```
csc /r:System.IO.Log.dll *.cs
```

Figure 17-13 shows the results of running the LoggerTestApp program one time. Running the program multiple times results in repeated log entries.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\Logging>loggertestapp
2/3/2008 12:23:46 PM Reactor Coolant 3 Main coolant pump speed limited
2/3/2008 12:23:46 PM Main Engine 3 Main condenser loss of vacuum
2/3/2008 12:23:46 PM Reactor Coolant 3 Main coolant pump speed limited
2/3/2008 12:23:46 PM Reactor 1 Loss of control rod control
2/3/2008 12:23:46 PM Reactor Coolant 3 Main coolant pump speed limited

C:\Documents and Settings\Rick\Desktop\Projects\Chapter17\Logging>_
```

Figure 17-13: Results of Running Example 17.19

Quick Review

The System.IO.Log namespace contains classes, structures, interfaces, and enumerations designed to help you create robust event logging services for your programs. Some of the functionality provided by the contents of the System.IO.Log namespace is only available on Microsoft Windows 2003r2 and Windows Vista or later operating systems. These operating systems come with the Common Log File System (CLFS).

Using FileDialogs

As you know by now, the .NET Framework provides a large collection of GUI components that make programming rich graphical user interfaces relatively painless. Most of these classes can be found in the System.Windows.Forms namespace. Two of those classes: OpenFileDialog and SaveFileDialog make it easy to graphically select and open or save files. The following example uses the OpenFileDialog class to select one or more files to open and display several file properties in a TextBox. The example consists of two classes: GUI and MainApp.cs.

17.20 GUI.cs

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4
5    public class GUI : Form {
6
7      private SplitContainer _splitContainer1;
8      private TextBox _textBox1;
9      private Button _button1;
10
11     public String TextBoxText {
12       get { return _textBox1.Text;  }
13       set { _textBox1.Text = value; }
14     }
15
16     public GUI(MainApp ma){
17        this.InitializeComponent(ma);
18     }
19
20     private void InitializeComponent(MainApp ma) {
21       _splitContainer1 = new SplitContainer();
22       _textBox1 = new TextBox();
23       _button1 = new Button();
24       _splitContainer1.Panel1.SuspendLayout();
25       _splitContainer1.Panel2.SuspendLayout();
```

 © 2008 Rick Miller — All Rights Reserved

```
26       _splitContainer1.SuspendLayout();
27       this.SuspendLayout();
28
29       _splitContainer1.Dock = DockStyle.Fill;
30       _splitContainer1.Location = new Point(0, 0);
31       _splitContainer1.Panel1.Controls.Add(_textBox1);
32       _splitContainer1.Panel2.Controls.Add(_button1);
33       _splitContainer1.Size = new Size(292, 273);
34       _splitContainer1.SplitterDistance = 161;
35       _splitContainer1.TabIndex = 0;
36
37       _textBox1.Location = new Point(3, 3);
38       _textBox1.AutoSize = true;
39       _textBox1.Anchor = (AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right);
40       _textBox1.Multiline = true;
41       _textBox1.Name = "textBox1";
42       _textBox1.Size = new Size(155, 267);
43       _textBox1.TabIndex = 0;
44
45       _button1.Location = new Point(27, 12);
46       _button1.Size = new System.Drawing.Size(75, 23);
47       _button1.TabIndex = 0;
48       _button1.Text = "Open File";
49       _button1.UseVisualStyleBackColor = true;
50       _button1.Click += new System.EventHandler(ma.Button1_Click);
51
52       this.AutoScaleMode = AutoScaleMode.None;
53       this.ClientSize = new System.Drawing.Size(292, 273);
54       this.Controls.Add(_splitContainer1);
55
56       this.Text = "FileDialog Demo";
57       _splitContainer1.Panel1.ResumeLayout(false);
58       _splitContainer1.Panel1.PerformLayout();
59       _splitContainer1.Panel2.ResumeLayout(false);
60       _splitContainer1.ResumeLayout(false);
61       this.ResumeLayout(false);
62     } // End InitializeComponent()
63   } // End class definition
```

Referring to Example 17.20 — the GUI class inherits from Form and uses a SplitContainer to hold a TextBox and a Button. The TextBox.MultiLine property is set to true and its Anchor property is set to anchor to all four sides of its containing panel. The button’s Click event is set to invoke the MainApp.Button1_Click() method.

17.21 MainApp.cs

```
1    using System;
2    using System.Windows.Forms;
3    using System.Text;
4    using System.IO;
5
6    public class MainApp {
7      private OpenFileDialog _fileDialog;
8      private GUI _gui;
9
10     public MainApp(){
11       _gui = new GUI(this);
12       _fileDialog = new OpenFileDialog();
13       _fileDialog.Multiselect = true;
14       Application.Run(_gui);
15     }
16
17     public void Button1_Click(Object o, EventArgs e){
18       _fileDialog.ShowDialog();
19       String[] filenames = _fileDialog.FileNames;
20       StringBuilder sb = new StringBuilder();
21       foreach(String s in filenames){
22         FileInfo file = new FileInfo(s);
23         sb.Append("FileName:" +  file.Name + "\r\n");
24         sb.Append("Directory:" + file.DirectoryName + "\r\n");
25         sb.Append("Size:" + file.Length + " Bytes\r\n");
26         sb.Append("\r\n");
27       }
28       _gui.TextBoxText = sb.ToString();
29     }
30
31     public static void Main(){
32       new MainApp();
33     }
34   } // end class definition
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 17.21 — the MainApp class plays host to the Main() method and the Button1_Click() event handler method. In the body of the MainApp constructor the OpenFileDialog object is created and it's Multiselect property is set to true. This allows the user to select multiple files to open at the same time.

When the button is clicked in the GUI, the Button1_Click() event handler method calls the OpenFileDialog's ShowDialog() method. This displays the dialog and lets users select the file(s) they wish to open. At this point the program effectively blocks until the user clicks the Open button on the OpenFileDialog window.

The OpenFileDialog.FileNames property returns a string array containing the names of the file(s) selected by the user. The `foreach` statement starting on line 21 iterates over each filename, creates a FileInfo object, extracts the required information about each file, and appends it to a StringBuilder object. When the `foreach` statement finishes, the file information contained in the StringBuilder object is written to the GUI.TextBoxText property, which in turn sets its TextBox's Text property.

Figure 17-14 shows the results of running this program and selecting three files named GUI.cs, MainApp.cs, and MainApp.exe. Your results will differ depending on what files you select.

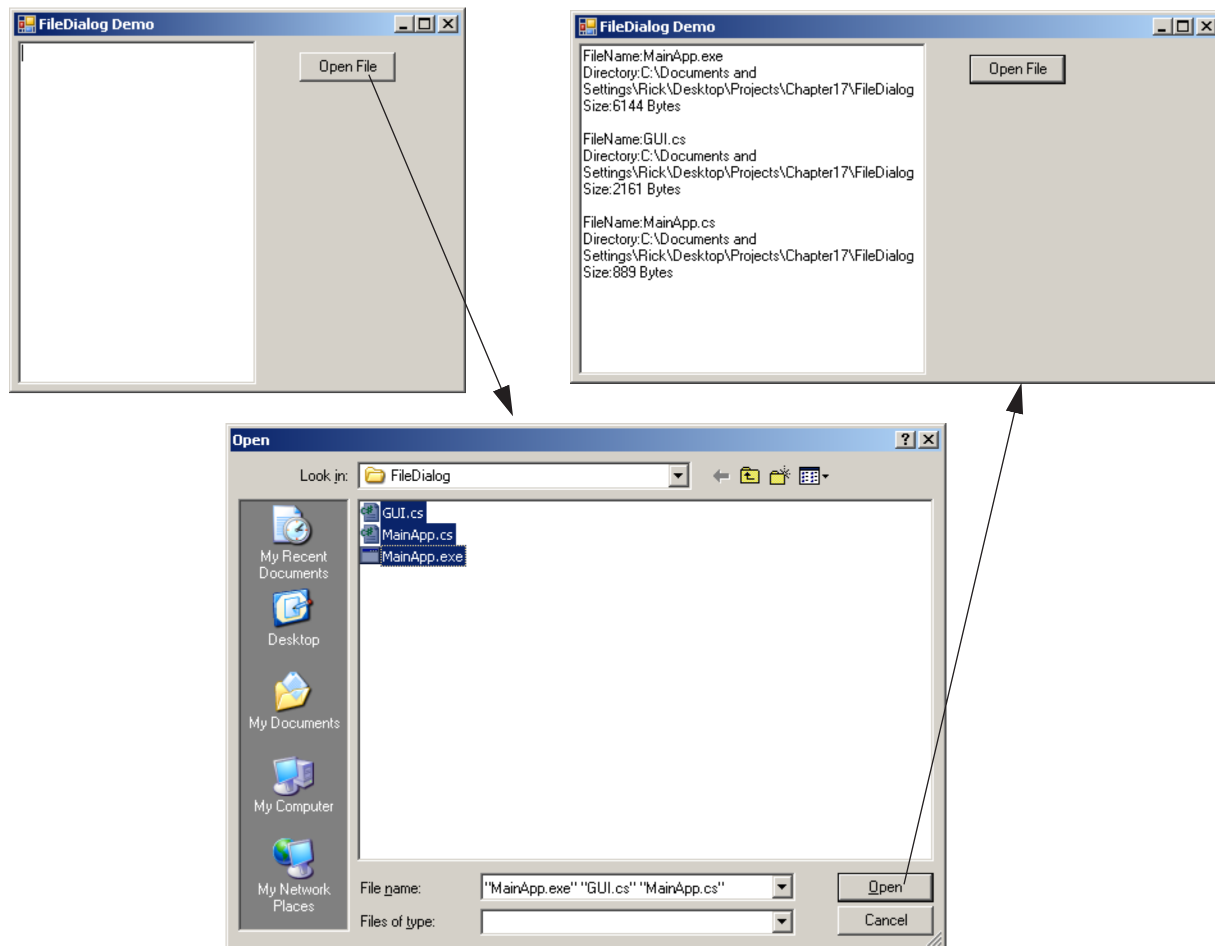


Figure 17-14: Results of Running Example 17.21 and Selecting Three Files

Quick Review

Use the OpenFileDialog and SaveFileDialog classes to graphically select and open/save files. The OpenFileDialog can be used to select multiple files simultaneously. When used in this manner, the OpenFileDialog.FileNames property returns a string array containing the names of the files selected.

 © 2008 Rick Miller — All Rights Reserved

Summary

In most all cases, data generated by an application and stored on an auxiliary storage device such as a hard disk, is saved as an organized, related collection of bits in a structure commonly referred to as a *file*.

It is the operating system's responsibility to manage the organization, reading, and writing of files. When a new storage device is added to your computer, it must first be formatted in a way that allows the operating system to access its data.

The file, from the operating system's point of view, is the fundamental storage organizational element. An application's associated data can be stored in one or more files. A file is located in another organizational element called a *directory*. A directory is a special type of file that contains a list of files and directories. A directory contained inside another directory is called a *subdirectory*.

The topmost directory structure is referred to as the *root* directory. The root directory of a particular drive is indicated by the name of the drive followed by a colon ':', followed by a backward slash character '\'. The root directory of the C drive would be "C:\".

The location of a particular file within a directory structure is indicated by a string of characters called a *path*. The path to the file's location can be *absolute* or *relative*. An *absolute path* includes the name or letter of the drive and all directory and subdirectory names required to pinpoint the file's location. A *relative path* is the path to a file from some arbitrary starting point, usually a working directory.

You can easily create and manipulate directories and files with the help of several classes provided in the .NET Framework System.IO namespace. These include the *Path*, *File*, *FileInfo*, *Directory*, *DirectoryInfo*, and *DriveInfo* classes.

Verbatim strings are formulated by preceding the string with the '@' character which signals the compiler to interpret the string literally, including special characters and line breaks.

Object serialization provides an easy, convenient way for you to persist application data to disk. Object serialization is also the least flexible way to store application data because you can't edit the resulting file. Use a FileStream object and a BinaryFormatter to serialize objects to disk. Before an object can be serialized it must be tagged as being serializable with the serializable attribute. Place the serializable attribute above the class declaration line.

When *serializing* a collection of objects, remember that all objects contained within the collection must be serializable. You don't have to worry about the collections themselves, including ordinary arrays, as they are already tagged as being serializable.

You can get around the limitation of ordinary serialization by serializing objects to disk in XML format. Use the StreamWriter and XMLSerializer classes to serialize objects to disk in XML format. Use a FileStream and XMLSerializer to deserialize objects from an XML file.

The StreamReader and StreamWriter classes let you read and write text files. Text files are usually processed line-by-line. Lines of text are terminated with the special characters *carriage-return and line-feed* (\r\n). Each line can contain one or more fields *delimited* by some character. The comma ',' is a commonly used field delimiter. Individual fields can be further delimited as required.

Look to the objects in your program to determine the type of information your text file(s) must contain. You'll need to save enough data to recreate objects.

Process a text file by reading each line and breaking it into *tokens* with the String.Split() method. If one or more fields are also delimited, use the String.Split() method to tokenize the data as required.

Use the BinaryReader and BinaryWriter classes to read and write binary data to disk. The BinaryWriter class provides an overloaded Write() method that is used to write each of the simple types including strings and arrays of bytes and characters. The BinaryReader class provides an assortment of Read*Typename*() methods where *Typename* may be any one of the simple types to include strings and arrays of bytes and characters.

You can conduct *random access file I/O* with the BinaryReader, BinaryWriter, and FileStream classes. The FileStream class provides a Seek() method that allows you to position the *file pointer* at any point within a file. As you learned in the previous section, the BinaryReader and BinaryWriter classes provide methods for reading and writing binary, string, byte, and character array data.

Use the OpenFileDialog and SaveFileDialog classes to graphically select and open/save files. The OpenFileDialog can be used to select multiple files simultaneously. When used in this manner the OpenFileDialog.FileNames property returns a string array containing the names of the files selected.

© 2008 Rick Miller — All Rights Reserved

Skill-Building Exercises

1. **API Drill:** Explore the contents of the System.IO namespace. List each entry and note its purpose.

2. **API Drill:** Explore the contents of the System.Runtime.Serialization and System.Runtime.Serialization.Formatters.Binary namespaces. List each entry and note its purpose.

3. **Programming Exercise:** Compile and run the examples in this chapter. Note their behavior. Experiment by making changes to each program to get different results.

4. **Create Sequence Diagrams:** Step through the code examples in this chapter and follow the paths of execution. Select a part of each program and create a detailed UML sequence diagram that shows objects used, method calls, and return values.

5. **API Drill:** Research the System.ArraySegment<T> generic class and note its purpose.

Suggested Projects

1. **Employee Database:** Write a GUI application that lets users create a database of employees. Use the Employee code given in Chapter 11. Users should be able to create employees and save their information to a file. Your application should have fields for entering employee information and some way of displaying a list of employees currently in the data base.

2. **Robot Rat:** Write a version of the robot rat program that records each movement the rat makes to disk. Create a feature called Auto-Playback that lets the robot rat read and execute a series of stored movements from a file.

3. **File Lister:** Write a GUI application that recursively traverses a directory and any subdirectories it might contain. Write to file a list of all the files contained within the directories along with any other data about each file users have selected from a set of menu options.

4. **Picture Display:** Write a GUI application that opens image files and displays their contents in a PictureBox.

5. **Asynchronous File I/O:** The FileStream class supports asynchronous file I/O with its BeginRead()/EndRead() and BeginWrite()/ EndWrite() methods. Research these methods and review asynchronous method calling in Chapter 16. Modify the Picture Display program described in suggested project 4 above to asynchronously read large image files.

Self-Test Questions

1. Before an object can be serialized, with what attribute must it be tagged?

2. (True/False) Before a collection of objects can be serialized, all objects contained within that collection must be serializable.

3. What must be done to a freshly deserialized object before being used in a program?

4. What three classes can be used together to perform random access file I/O?

 © 2008 Rick Miller — All Rights Reserved

5. What's the difference between the File class and the FileInfo class?

6. What's the difference between the Directory class and the DirectoryInfo class?

7. What must be done to a new storage device before the computer can use it to read and write data?

8. Describe in your own words the definition of the term *file*.

9. What term is used to describe the topmost directory?

10. Another word that's synonymous with directory is ________________.

11. The location of a particular file within a directory structure is indicated by a string of characters called a ______.

12. What's the difference between an absolute path and a relative path?

13. What's the advantage of using a verbatim string to formulate file paths?

14. A character used to separate individual fields in a text file record is called a _______________.

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

Notes

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Part V: Network & Database Programming

© 2008 Rick Miller — All Rights Reserved

 © 2008 Rick Miller — All Rights Reserved

Chapter 18

Contax T / Kodak Tri-X

Gerogetown Walkabout

Network Programming Fundamentals

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 server and client applications can be physically and logically distributed*
- *Describe the properties of a multitiered application*
- *State the definition of the terms "protocol", "port", "packet", "datagram", "TCP/IP" and "UDP"*
- *List the requirements for testing both client and server applications on one computer*

© 2008 Rick Miller — All Rights Reserved

Introduction

Network applications pervade today's modern computing environment. If you use email, a web browser, or a chat program like Windows Live Messenger, you're 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*, *packet*, *datagram*, *TCP/IP* and *UDP*, and how applications can be physically and logically distributed in a networked environment.

The second purpose of this chapter is to introduce you to the concepts of *multitiered*, *distributed* applications. Modern network applications are often *logically tiered*, with one or more of their 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 applications.

Upon completion of this chapter, you will have a solid foundation upon which to successfully approach *Chapter 19 - Networked Client-Server Applications*, and *Chapter 20 - Database Access & Multitiered Applications*. 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, 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 18-1 shows a diagram for a simple computer network. This type of simple network is referred to as a *local area network* (LAN).

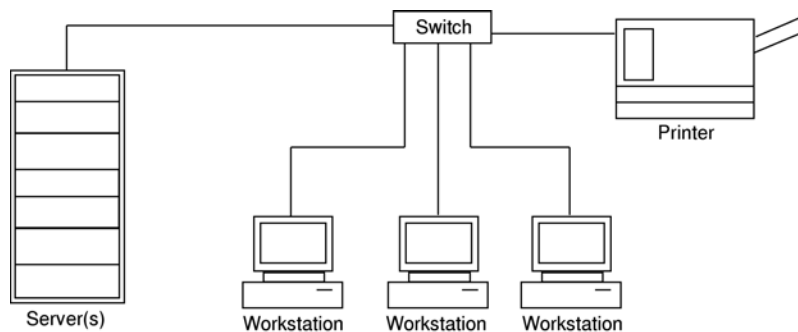


Figure 18-1: A Simple Computer Network

Referring to Figure 18-1 — 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 work-

 © 2008 Rick Miller — All Rights Reserved

stations 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 or adhere to 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. Apple Macs 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 18-2 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.

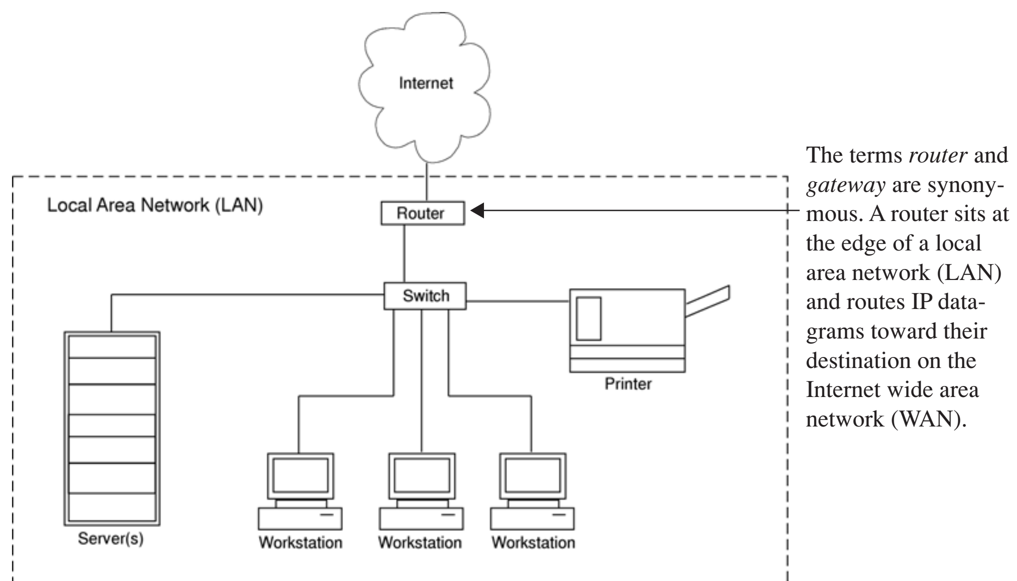


Figure 18-2: Local Area Network Connected to the Internet

© 2008 Rick Miller — All Rights Reserved

What's So Special About The Internet?

What makes the Internet so special? The answer is — TCP/IP. The Internet is a vast network of computer networks. All of 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 one packet at a time to the designated computer. TCP/IP provides *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 18-3 shows how the simple network of Agency A can share resources with other agencies via the Internet.

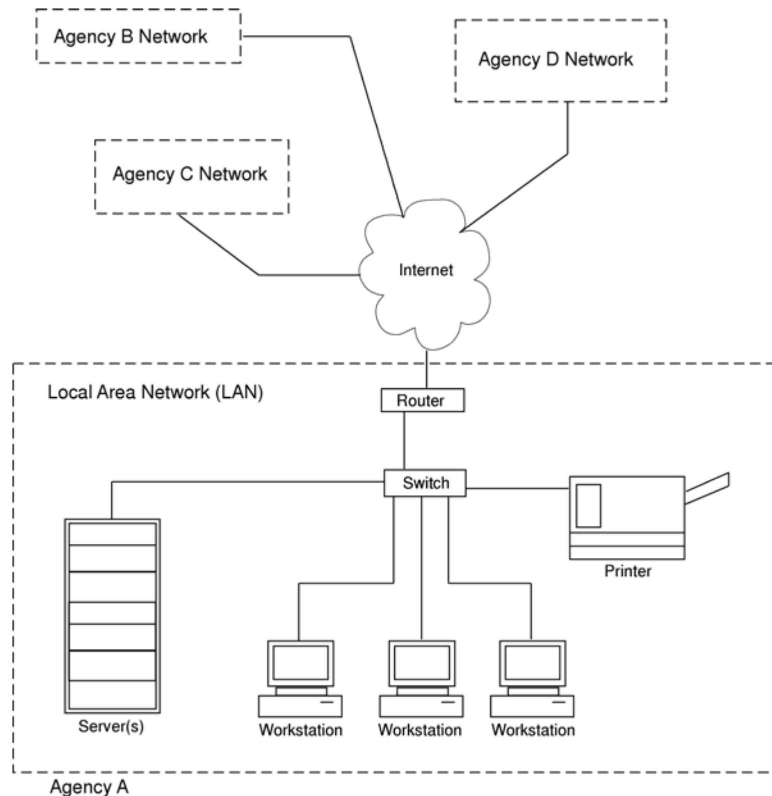


Figure 18-3: The Internet — A Network of Networks Communicating via Internet Protocols

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, 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 or adhere to 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.

© 2008 Rick Miller — All Rights Reserved

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 connotation. 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 18-4 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.

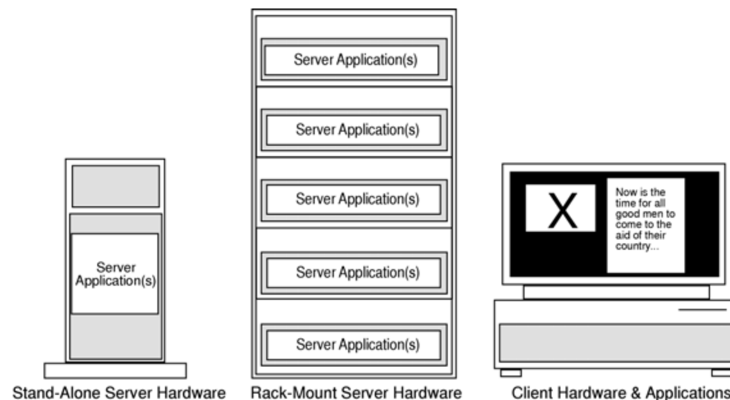


Figure 18-4: Client and Server Hardware and Applications

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.

© 2008 Rick Miller — All Rights Reserved

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 multilayered 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 client-server applications in Chapter 19, 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 18-5 illustrates the concept of running client and server applications on the same physical hardware.

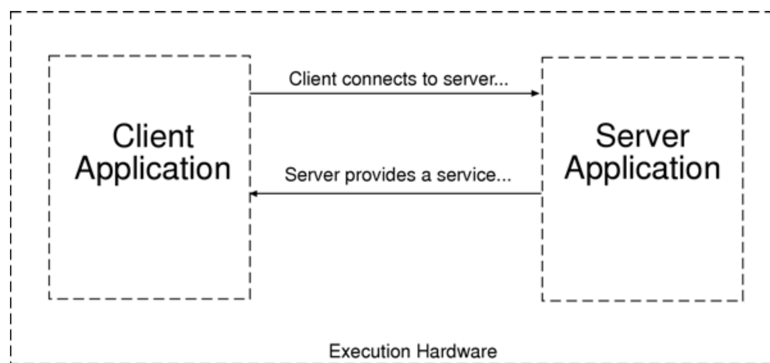


Figure 18-5: Client and Server Applications Physically Deployed to the Same Computer

Running Multiple Clients On The Same Computer

You can run multiple client applications on the same computer. To do this, 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 *multithreaded*. 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 18-6 illustrates the concept of running multiple client applications and the server application on the same hardware.

 © 2008 Rick Miller — All Rights Reserved

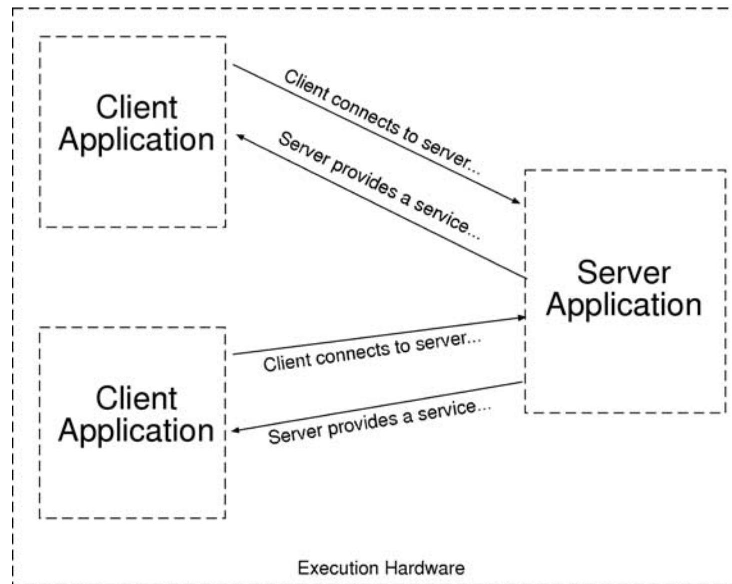


Figure 18-6: Running Multiple Clients on Same Hardware

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 18-7 illustrates this concept.

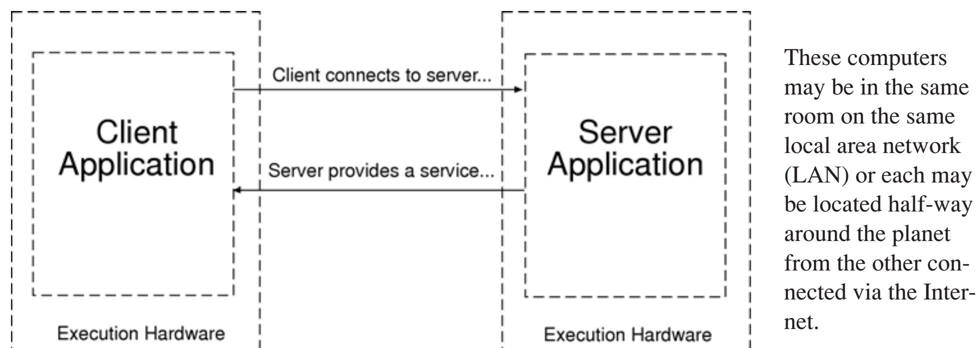


Figure 18-7: Client and Server Applications Deployed on Different Computers

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 or to different physical computers. These computers may be in the same room or located a great distance from each other.

© 2008 Rick Miller — All Rights Reserved

Multitiered Applications

Up until 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 composed of more than one tier is referred to as a *multitiered* application. This section discusses the concepts related to multitiered applications in greater detail.

Logical Application Tiers

Figure 18-8 illustrates the concept of a multitiered application.

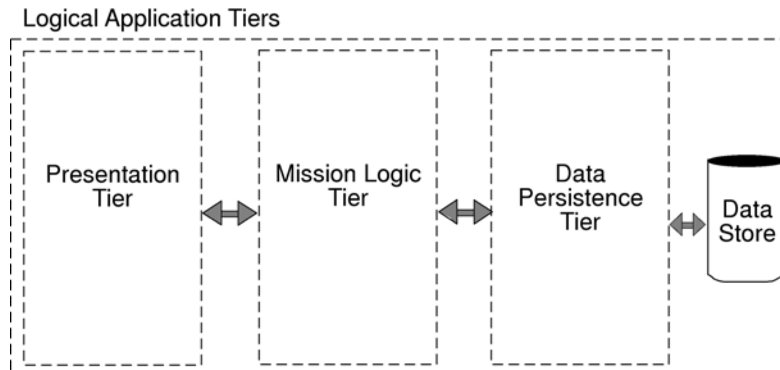


Figure 18-8: A Multitiered Application

Referring to Figure 18-8 — in this example the application comprises 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.) (I use the term mission logic tier interchangeably with the term business logic tier when referring to multitiered applications written for Department of Defense clients.) The data persistence tier is responsible for servicing the data needs (*i.e.,* data 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 18-9.

It is more likely the case, however, that logical application tiers are physically deployed to separate and distinct computing nodes located some distance apart. Figure 18-10 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

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 *multitiered 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,

© 2008 Rick Miller — All Rights Reserved

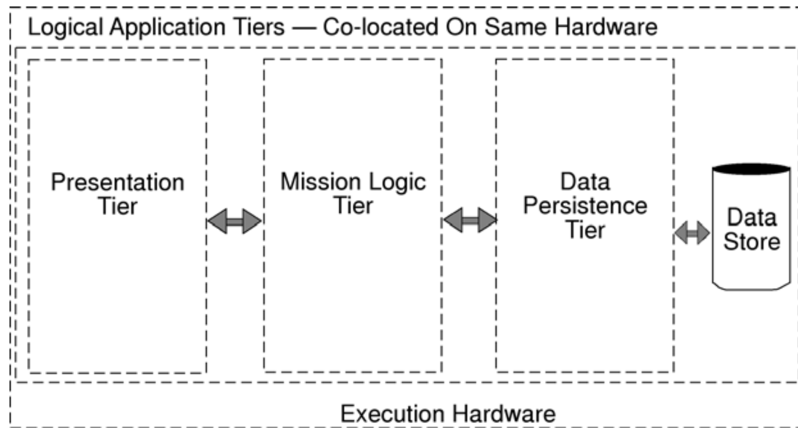


Figure 18-9: Physically Deploying Logical Application Tiers on Same Computer

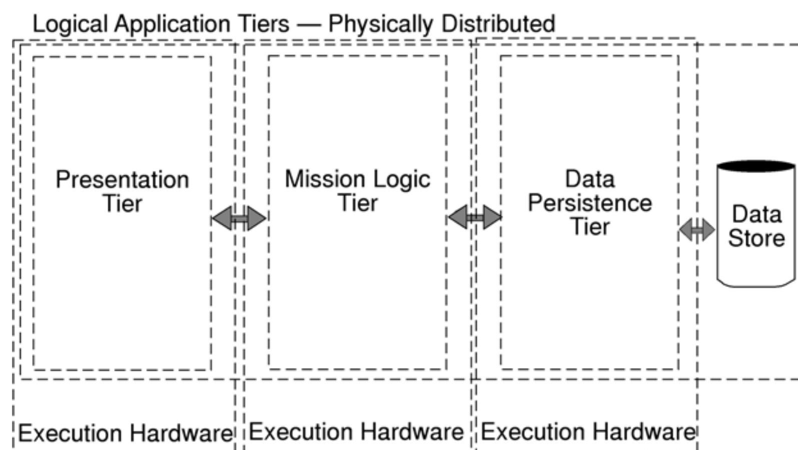


Figure 18-10: Logical Application Tiers Physically Deployed to Different Computers

and 3) the *data persistence tier*, which is concerned with the quick and reliable delivery of data to the mission logic tier. A multitiered 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 System.Net namespace 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 are routed through remaining network connections to their intended destination.

The Internet protocols work together as a layered *protocol stack* as is shown in Figure 18-11.

 © 2008 Rick Miller — All Rights Reserved

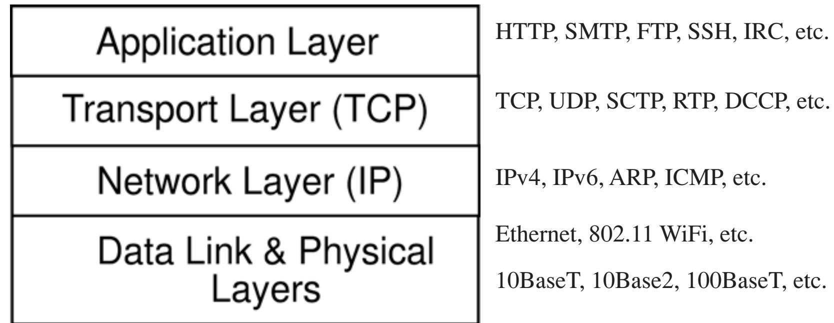


Figure 18-11: TCP/IP Protocol Stack

The layered protocol stack consists of the *application layer*, the *transport layer*, the *network layer*, the *data link layer*, and the *physical layer*. Each protocol stack layer 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 18-11. 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). The .NET Framework 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.)

 © 2008 Rick Miller — All Rights Reserved

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 may 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 And Physical Layers

The *data link* and *physical* layers are the lowest layers of the networking protocol stack. It is here that data is placed "on the wire" for transmission across the LAN or across the world.

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 18-12.

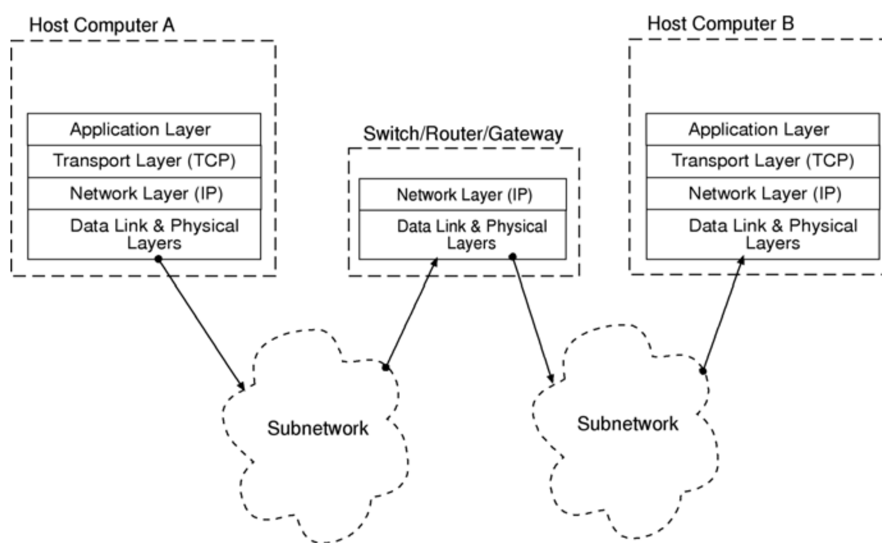


Figure 18-12: Internet Protocol Stack Operations

© 2008 Rick Miller — All Rights Reserved

Referring to Figure 18-12 — 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, then 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. The .NET Framework provides a set of classes in the System.Net namespace that makes network programming easy.

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.

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 (PDAs), 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

 © 2008 Rick Miller — All Rights Reserved

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, but most likely they are deployed to different machines.

Client and server applications can be logically separated into distinct functional areas called *tiers*. Applications logically segmented in this fashion are referred to as *multitiered 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. A multitiered 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 routing 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. 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.

Skill-Building Exercises

1. **Web Research:** Expand your understanding of the TCP/IP protocols. Search the web for the Internet RFCs used as references for this chapter.

2. **Web Research:** Expand your understanding of network applications. Search the web for material related to packet-

 © 2008 Rick Miller — All Rights Reserved

switched networks, distributed applications, and multitiered applications.

Suggested Projects

1. None

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?

References

RFC 791 - Internet Protocol

RFC 2396 - Uniform Resource Identifiers (URI): General Syntax

RFC 793 - Transmission Control Protocol

RFC 768 - User Datagram Protocol

Uyless Black. *Advanced Internet Technologies*. Prentice Hall Series In Advanced Communications Technologies. Prentice Hall PTR, Upper Saddle River, NJ. ISBN: 0-13-759515-8

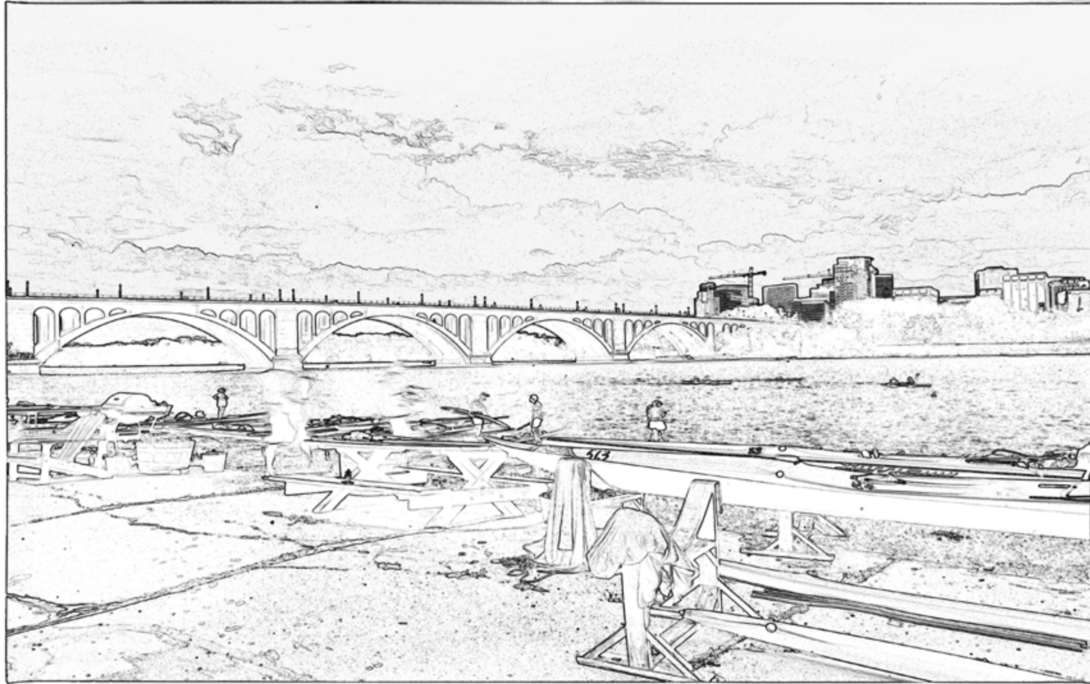
 © 2008 Rick Miller — All Rights Reserved

Notes

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Chapter 19



Pentax 67 / SMC Takumar 55/2.8 / Kodak Tri-X Professional

Rosslyn, VA & Key Bridge From Washington Canoe Club

Networked Client-Server Applications

Learning Objectives

- *Demonstrate your ability to create networked client-server applications*
- *Utilize .NET Remoting to create client-server applications*
- *State the three things you must do to create a .NET Remoting application*
- *Create .NET Remoting applications with and without configuration files*
- *Create .NET Remoting applications using remote object interfaces*
- *Create client-server applications using the TcpListener and TcpClient classes*
- *Create a multithreaded TCP/IP server application*
- *Use a ParameterizedThreadStart delegate method to create a multithreaded client-server application*
- *Create a TCP/IP client application*
- *Send serialized objects between networked client-server applications*
- *Create a custom application protocol for use in a client-server application*
- *Use StreamReader and StreamWriter objects to send and receive data in a client-server application*
- *Use the NetworkStream to serialize/deserialize objects between client-server applications*

© 2008 Rick Miller — All Rights Reserved

Introduction

You're going to have a lot of fun in this chapter. It is here that you'll put into practical use many of the networking concepts discussed in the previous chapter.

You will start by learning how to build client-server applications using Microsoft's .NET remoting technology. .NET remoting makes it possible to utilize the services of an object hosted on a remote computer or locally across applications boundaries. I'll show you how to create .NET remoting applications using programmatic configuration and configuration files. I'll also show you how to call the services of a remote object via one of its implemented interfaces.

Once you've mastered the art of .NET remoting, you'll learn how to create client-server applications using the TcpListener, TcpClient, and other classes found in the System.Net namespace. I'll show you how to create a multithreaded server application that can service requests from multiple clients. I'll also show you how to create a custom application protocol so your client-server applications can talk to each other.

If you are timid about the thought of building a networked application, don't be. I will show you what to do and how to compile your project files every step of the way. When you've finished this chapter, you will have the confidence to build your own networked client-server applications.

Also, what you learn here will be put to good use in *Chapter 20: Database Access & Multitiered Applications*.

Building Client-Server Applications With .NET Remoting

.NET remoting makes it possible to access the services of an object hosted locally but in a different application domain or hosted remotely on another computer located somewhere in network land. The .NET remoting infrastructure hides many of the nasty details normally associated with network programming, letting you focus on building value-added applications. You'll find .NET remoting to be an easy and powerful way to build client-server applications.

The Three Required Components of A .NET Remoting Application

All .NET remoting applications have three common components regardless of their complexity: a *remotable object*, a *server application* that hosts the remotable object and handles incoming service requests, and a *client application* that utilizes the services of the remotely-hosted object. Figure 19-1 illustrates these concepts.

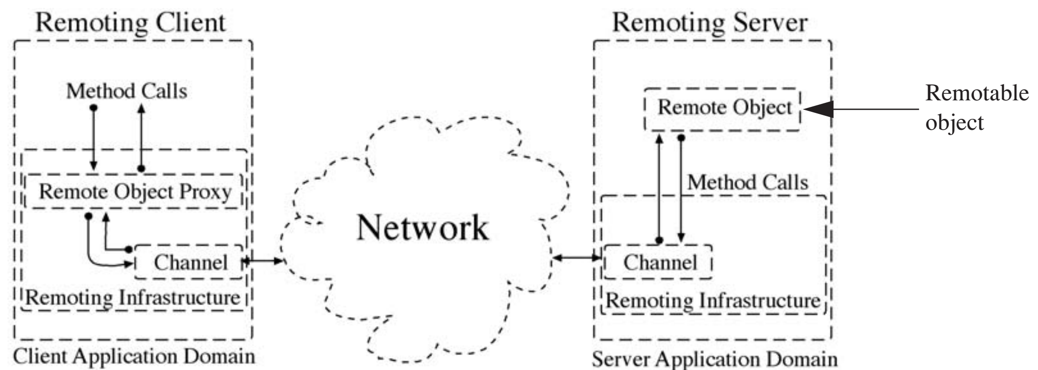


Figure 19-1: .NET Remoting Architecture

Referring to Figure 19-1 — the three components of a typical .NET remoting application include a remotable object, a server application that coordinates method calls to the remote object and makes its services available on a particular channel, and a client application that accesses the services of the remote object via the appropriate channel.

A remotable object is created from a class that inherits from System.MarshalByRefObject. This enables the object to be shared across application domains in .NET remoting applications. Note that the remotable object can be

 © 2008 Rick Miller — All Rights Reserved

simple or complex. In this section, I'll keep the remotable objects simple, but at the end of this section you'll see how a remote object can serve as a façade to a complex data-driven application.

The remoting server application hosts the remotable object and makes its services available via a channel. There are three primary channel types: *TcpChannel*, *HttpChannel*, and *IpcChannel*. The IpcChannel is used for *inter-process* communication between client and server applications hosted on the same machine.

The remoting client application accesses the services of the remote object via a *proxy* created by the .NET remoting infrastructure. The .NET remoting infrastructure is responsible for creating the proxy object, setting up and tearing down network communications between client and server applications, and marshaling and unmarshaling remote method calls and any returned objects between client and server. All this is done under the covers as you will soon see.

A Simple .NET Remoting Application

In this section I'll show you how to create a simple .NET remoting application. It all starts with the creation of a remotable class type you can use to create remotable objects. Example 19.1 gives the code for a class named TestClass.

19.1 TestClass.cs

```
1    using System;
2
3    public class TestClass : MarshalByRefObject{
4
5      private string _text;
6
7      public string Text {
8        get { return _text; }
9        set {
10         _text = value;
11         Console.WriteLine("Property changed --> " + _text);
12         }
13     }
14
15     public TestClass():this("This is the default text message!"){}
16
17     public TestClass(string s){
18       _text = s;
19     }
20   }
```

Referring to Example 19.1 — TestClass extends System.MarshalByRefObject. This tags objects of type TestClass as being remotable. TestClass has two constructors and one property named Text. Setting the Text property causes a short message to be written to the console. This is *not* normally a good thing to do in a property but in this case it will help to demonstrate some important remoting concepts.

Compile TestClass into a dynamically linked library (dll) by issuing the following compiler command at the command line:

```
csc /t:library TestClass.cs
```

The reason you need to compile this into a dll is that you'll need to share this code with both the server and client applications.

Next, let's create the server application that will host an instance of TestClass. Example 19.2 gives the code for a class named RemotingServer.

19.2 RemotingServer.cs

```
1    using System;
2    using System.Runtime.Remoting;
3    using System.Runtime.Remoting.Channels;
4    using System.Runtime.Remoting.Channels.Tcp;
5
6    public class RemotingServer {
7      public static void Main(){
8
9        try{
10       TcpChannel channel = new TcpChannel(8080);
11       ChannelServices.RegisterChannel(channel, false);
12       RemotingConfiguration.RegisterWellKnownServiceType(typeof(TestClass), "TestClass",
13                                                     WellKnownObjectMode.SingleCall);
14       Console.WriteLine("Listening for remote requests. Press any key to exit...");
15       Console.ReadLine();
16       }catch(ArgumentNullException ane){
17         Console.WriteLine("Channel argument was null!");
```

 © 2008 Rick Miller — All Rights Reserved

```
18          Console.WriteLine(ane);
19        }catch(RemotingException re){
20          Console.WriteLine("Channel has already been registered!");
21          Console.WriteLine(re);
22        }catch(Exception e){
23          Console.WriteLine(e);
24        }
25      } // end Main()
26    } // end class definition
```

Referring to Example 19.2 — notice first the list of namespaces you must rely upon. These include System.Runtime.Remoting, System.Runtime.Remoting.Channels, and System.Runtime.Remoting.Channels.Tcp. I am using the System.Runtime.Remoting.Channels.Tcp namespace because I'm going to make available the services of a TestClass object via a TcpChannel.

The first thing the server does is create the TcpChannel object to listen on port 8080. Be sure this port is not in use or the TcpChannel() constructor call will throw an exception.

Next, on line 11, the newly created channel is registered with the help of the ChannelServices.RegisterChannel() method. The second argument to the RegisterChannel() method specifies whether or not to enforce security on the channel. In this case I have opted not to enforce security by supplying a value of false.

The meat of the server comes on line 12 where an object of type TestClass is registered. The RemotingConfiguration.RegisterWellKnownServiceType() method takes three arguments: The first is the *type* of the object to be hosted, the second is a string indicating the *service name* by which the object can be accessed, and the third argument specifies whether calls to the remote object's methods are handled by a new instance of the object (*SingleCall*) or by one object that persists across multiple service requests (*Singleton*). In this example, I am using the SingleCall mode.

To compile this program I recommend putting the TestClass.dll in the same directory as the RemotingServer.cs class code and issuing the following compiler command:

```
csc /r:TestClass.dll RemotingServer.cs
```

The `/r` switch tells the compiler to include the indicated resource during compilation. When you've compiled the server give it a whirl. Figure 19-2 shows the RemotingServer running and waiting for an incoming connection.

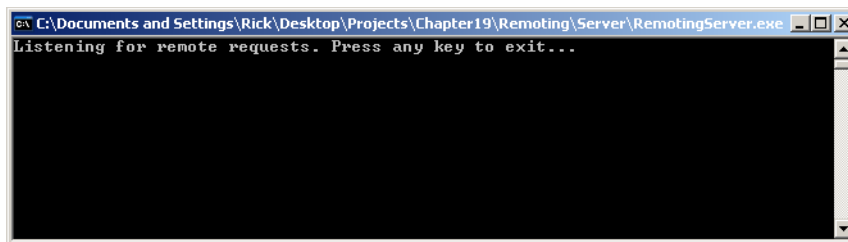


Figure 19-2: RemotingServer Waiting for Something to do

Finally, we need a client application that makes calls to the remote TestClass object hosted on the running RemotingServer application. Example 19.3 gives the code for the RemotingClient application.

19.3 RemotingClient.cs

```
1     using System;
2     using System.Runtime.Remoting;
3     using System.Runtime.Remoting.Channels;
4     using System.Runtime.Remoting.Channels.Tcp;
5
6     public class RemotingClient {
7       public static void Main(){
8         try {
9         TcpChannel channel = new TcpChannel();
10        ChannelServices.RegisterChannel(channel, false);
11        TestClass test = (TestClass)Activator.GetObject(typeof(TestClass), "tcp://localhost:8080/TestClass" );
12        Console.WriteLine(test.Text);
13        test.Text = "This is a new string sent from the client application!";
14        Console.WriteLine(test.Text);
15        }catch(ArgumentNullException ane){
16          Console.WriteLine(ane);
17        }catch(RemotingException re){
18          Console.WriteLine(re);
19        }catch(Exception e){
20          Console.WriteLine(e);
21        }
22      }
23    }
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 19.3 — on lines 9 and 10, a TcpChannel object is created and registered. The heavy lifting occurs on line 11 when an instance of the remote TestClass object is created with the help of the System.Activator.GetObject() method. This method takes two arguments: the *type* of object to create, and a string representing the *URL of the remote service*. In this example, as you'll recall from the RemotingServer code, the remote TestClass object is hosted on the server and is made available via a service named "TestClass" on port 8080. Thus, the URL given here must reflect that reality. Notice that the retrieved object must be cast to its proper type. Once this line of code executes, the client application can use the reference *test* as though the object it pointed to was on the local machine. All the network calls to the remote object are handled automatically by the .NET remoting infrastructure.

In this example, on line 12, I write the value of the Text property to the console. I then attempt to change the Text property by setting it to a new string value.

To compile this program, you'll need to make a copy of the TestClass.dll, put it in the same folder where you have the RemotingClient.cs code and issue the following compiler command:

```
csc /r:TestClass.dll RemotingClient.cs
```

Figure 19-3 shows the results of running the RemotingClient application several times. Make sure to start the RemotingServer before running RemotingClient. Also, I have assumed you're testing this application on the local machine (localhost). If you have two computers on a network, change localhost to the appropriate IP address and recompile the programs before you start the server and run the client application.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\Remoting\Server\RemotingServer.exe
Listening for remote requests. Press any key to exit...
Property changed --> This is a new string sent from the client application!
Property changed --> This is a new string sent from the client application!
Property changed --> This is a new string sent from the client application!
Property changed --> This is a new string sent from the client application!
```

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\Remoting\Client>remotingclient
This is the default text message!
This is the default text message!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\Remoting\Client>remotingclient
This is the default text message!
This is the default text message!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\Remoting\Client>remotingclient
This is the default text message!
This is the default text message!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\Remoting\Client>remotingclient
This is the default text message!
This is the default text message!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\Remoting\Client>_
```

Figure 19-3: Results of Running RemotingServer and RemotingClient with a SingleCall Mode Remote Object

SingleCall vs. Singleton

Referring to Figure 19-3 — notice that each time the RemotingClient application executes it gets the original, or default, remote object's Text property message, even though it sets the remote object's Text property to a new value. This is because all requests to the remote object are handled by a new object instance. (Note that the remote object's Text property is being changed by examining the RemotingServer's console output.) This behavior was set when I registered the remote object in the server code by using the WellKnownObjectMode.*SingleCall* mode.

If you want the remote object to persist and maintain state between client service requests, use the WellKnownObjectMode.*Singleton* mode. Example 19.4 gives the code for a slightly modified version of RemotingServer that registers the TestClass object in Singleton mode.

19.4 RemotingServer.cs (Mod 1)

```
1     using System;
2     using System.Runtime.Remoting;
3     using System.Runtime.Remoting.Channels;
4     using System.Runtime.Remoting.Channels.Tcp;
5
6     public class RemotingServer {
7       public static void Main(){
8
```

 © 2008 Rick Miller — All Rights Reserved

```
9        try{
10       TcpChannel channel = new TcpChannel(8080);
11       ChannelServices.RegisterChannel(channel, false);
12       RemotingConfiguration.RegisterWellKnownServiceType(typeof(TestClass), "TestClass",
13                                                    WellKnownObjectMode.Singleton);
14       Console.WriteLine("Listening for remote requests. Press any key to exit...");
15       Console.ReadLine();
16       }catch(ArgumentNullException ane){
17         Console.WriteLine("Channel argument was null!");
18         Console.WriteLine(ane);
19       }catch(RemotingException re){
20         Console.WriteLine("Channel has already been registered!");
21         Console.WriteLine(re);
22       }catch(Exception e){
23         Console.WriteLine(e);
24       }
25
26     }
27   }
```

Referring to Example 19.4 — the only difference between this and the previous version of RemotingServer appears on line 13 where I've registered the TestClass object in the WellKnownObjectMode.Singleton mode. The TestClass and RemotingClient code remain unchanged. Recompile the RemotingServer class and restart the server. Figure 19-4 shows the results of running the RemotingClient application several times. Note the different behavior. The remote object's Text property persists across service requests.

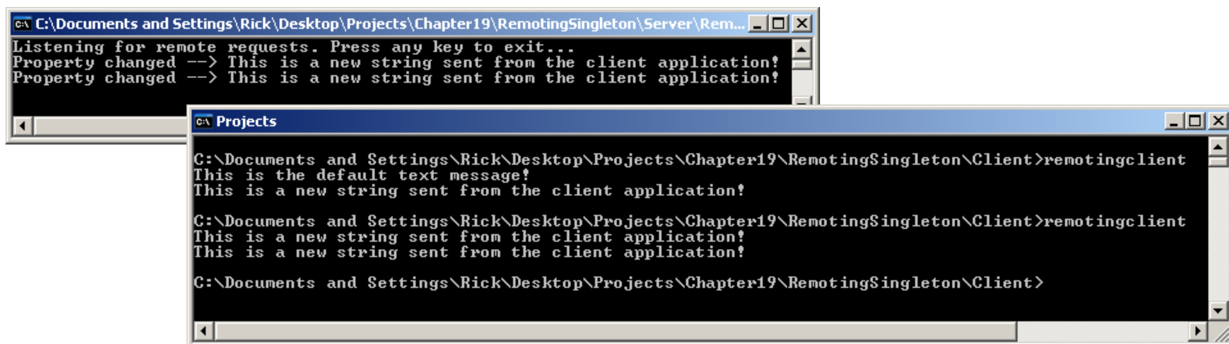


Figure 19-4: Results of Hosting TestClass Remote Object in Singleton Mode

Accessing A Remote Object Via An Interface

In the previous remoting examples, the TestClass.dll file was deployed with both the client and server applications. This is required because you're accessing a class by name and the compiler must resolve that name by having access to the code where that name is defined.

As it turns out, because the .NET remoting infrastructure creates a proxy to the remote object, the proxy is, in effect, only an interface to the remote object. (The .NET remoting infrastructure builds the proxy from the TestClass definition contained within the TestClass.dll.) You can make internal changes to TestClass, recompile it, and deploy the dll on the server side, and the client will still run fine. If, however, you wanted to swap out the TestClass remote object with a remote object of a different type, you'd have to deploy the new dll to both the client and server applications and recompile them both. There is a better way — have TestClass implement an interface and deploy the interface to the remote client. That way, you can change the type of remote object anytime you need to, so long as it implements the interface expected by the remote client.

Let's see how this is done. I'm going to make a few changes to the code base. It all begins with the definition of an interface I'll call ITest, which is given in Example 19.5.

19.5 ITest.cs

```
1   using System;
2
3   public interface ITest {
4     string Text{
5       get;
6       set;
7     }
8   }
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 19.5 — the ITest interface is quite simple. It simply declares the read/write property named Text. Compile this interface into a dll using the following compiler command:

```
csc /t:library ITest.cs
```

Example 19.6 gives the code for the modified TestClass.

19.6 TestClass.cs (Mod 1)

```
1    using System;
2
3    public class TestClass : MarshalByRefObject, ITest {
4
5      private string _text;
6
7      public string Text {
8        get { return _text; }
9        set {
10          _text = value;
11          Console.WriteLine("Property changed --> " + _text);
12          }
13     }
14
15     public TestClass():this("This is the new default text message!"){}
16
17     public TestClass(string s){
18       _text = s;
19     }
20   }
```

Referring to Example 19.6 — note now on line 3 that TestClass, in addition to extending MarshalByRefObject, implements the ITest interface. That's the only change to the TestClass code. Compile TestClass into a dll by using the following compiler command:

```
csc /t:library /r:ITest.dll TestClass.cs
```

Now, the RemotingServer code remains unchanged. The only thing you must do at this point is recompile RemotingServer and include a reference to both the TestClass.dll and ITest.dll files like so:

```
csc /r:TestClass.dll;ITest.dll RemotingServer.cs
```

The biggest changes are made to the RemotingClient application. Its code is given in Example 19.7.

19.7 RemotingClient.cs (Mod 1)

```
1    using System;
2    using System.Runtime.Remoting;
3    using System.Runtime.Remoting.Channels;
4    using System.Runtime.Remoting.Channels.Tcp;
5
6    public class RemotingClient {
7      public static void Main(){
8       try {
9        TcpChannel channel = new TcpChannel();
10       ChannelServices.RegisterChannel(channel, false);
11       ITest test = (ITest)Activator.GetObject(typeof(ITest), "tcp://localhost:8080/TestClass" );
12       Console.WriteLine(test.Text);
13       test.Text = "This is a new string sent from the client application";
14       Console.WriteLine(test.Text);
15       }catch(ArgumentNullException ane){
16         Console.WriteLine("Channel argument was null!");
17         Console.WriteLine(ane);
18       }catch(RemotingException re){
19         Console.WriteLine("Channel has already been registered!");
20         Console.WriteLine(re);
21       }catch(Exception e){
22         Console.WriteLine(e);
23       }
24     }
25   }
```

Referring to Example 19.7 — note the changes made to line 11. The RemotingClient application is getting an instance of a remote object of type ITest. Also note that the URL to the remote object remains unchanged, and this is fine. What matters in the code is that you're referencing ITest, not TestClass. To compile this program you'll need to copy the ITest.dll file to the client directory (and delete the TestClass.dll) and use the following compiler command:

```
csc /r:ITest.dll RemotingClient.cs
```

Now, start up the server and run the RemotingClient application several times. You'll see that outwardly it behaves like the previous version of the program as is shown in Figure 19-5. Inwardly, however, you've built yourself a .NET remoting application that can more flexibly respond to object changes on the back-end. And we're going to take this flexibility one step further in the following section when I show you how to configure both the server and client applications with configuration files.

© 2008 Rick Miller — All Rights Reserved

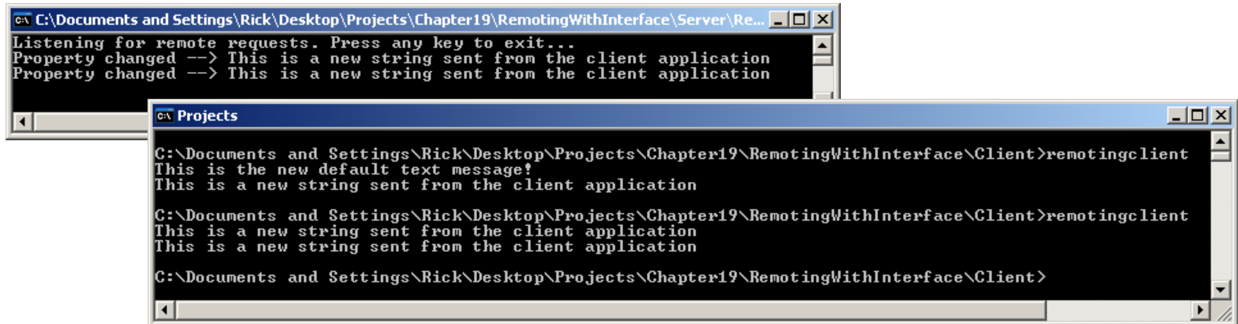


Figure 19-5: Results of Accessing a Remote Object via an Interface

Using Configuration Files

Both remote client and server applications can be configured via configuration files. A configuration file contains information about channels, remote object types, service names, etc. Using configuration files simplifies code and increases application flexibility by eliminating the need to recompile code when you want to make simple changes to certain application properties.

The use of configuration files requires changes to both the client and server remoting application code. The code for ITest and TestClass remains unchanged. Example 19.8 gives the code for the RemotingServer application modified to use a configuration file.

19.8 RemotingServer.cs (Mod 2)

```
1     using System;
2     using System.Runtime.Remoting;
3     using System.Runtime.Remoting.Channels;
4     using System.Runtime.Remoting.Channels.Tcp;
5
6     public class RemotingServer {
7       public static void Main(){
8        try {
9          RemotingConfiguration.Configure("server.config", false);
10         Console.WriteLine("Listening for remote requests. Press any key to exit...");
11         Console.ReadLine();
12        }catch(Exception e){
13          Console.WriteLine(e);
14        }
15      }
16    }
```

Referring to Example 19.8 — the RemotingConfiguration.Configure() method specifies a configuration file named server.config. The content of the server.config configuration file is listed in Example 19.9.

19.9 server.config

```
1     <configuration>
2       <system.runtime.remoting>
3         <application>
4           <service>
5            <wellknown mode="Singleton" type="TestClass, TestClass" objectUri="TestClass" />
6           </service>
7           <channels>
8             <channel ref="tcp" port="8080" />
9           </channels>
10        </application>
11      </system.runtime.remoting>
12    </configuration>
```

Referring to Example 19.9 — the server.config file contains XML tags that represent remoting configuration settings. (A full description of the remoting configuration file schema can be found on the MSDN website.) Configuration settings for one or more services hosted on one or more channels appear within the <application></application> tags. In this example the TestClass remote object type is hosted on a TcpChannel on port 8080 with a service URI named TestClass. Note on line 5 the type attribute is set to “TestClass, TestClass”. The first TestClass refers to the type name; the second TestClass refers to the application domain where TestClass can be found. (The typename and application domain are the same in this case.)

Use the following compiler command to compile the RemotingServer code:

```
csc /r:TestClass.dll;ITest.dll RemotingServer.cs
```

 © 2008 Rick Miller — All Rights Reserved

To run the RemotingServer application, make sure the server.config file is located in the same directory. (Otherwise, provide an absolute path name to the server.config file when you specify it in the call to the Configure() method.)

I made a few changes to the RemotingClient application as well. The modified code appears in Example 19.10.

19.10 RemotingClient.cs

```
1    using System;
2    using System.Runtime.Remoting;
3    using System.Runtime.Remoting.Channels;
4    using System.Runtime.Remoting.Channels.Tcp;
5
6    public class RemotingClient {
7      public static void Main(){
8       try {
9         RemotingConfiguration.Configure("client.config", false);
10        WellKnownClientTypeEntry[] client_types = RemotingConfiguration.GetRegisteredWellKnownClientTypes();
11        ITest test = (ITest)Activator.GetObject(typeof(ITest), client_types[0].ObjectUrl );
12        Console.WriteLine(test.Text);
13        test.Text = "This is a new string sent from the client application";
14        Console.WriteLine(test.Text);
15       }catch(Exception e){
16         Console.WriteLine(e);
17       }
18     }
19   }
```

Referring to Example 19.10 — the RemotingConfiguration.Configure() method is used to load a configuration file named client.config. Now, for maximum flexibility, on line 10, I have used the GetRegisteredWellKnowClientTypes() method to retrieve an array of registered client types. (In this example there is only one, as you'll see when you examine the client.config file.) I then use the client_types array to access the URL for the first registered client type, which in this case refers to the TestClass remote object service hosted on localhost port 8080. By doing this, and using the Activator.GetObject() method, I can change the client to access different remote objects without recompiling, provided those remote objects implement the ITest interface.

Example 19.11 lists the content of the client.config file.

19.11 client.config

```
1    <configuration>
2      <system.runtime.remoting>
3        <application>
4          <client>
5            <wellknown type="ITest, ITest" url="tcp://localhost:8080/TestClass" />
6          </client>
7        </application>
8      </system.runtime.remoting>
9    </configuration>
```

Use the following compiler command to compile the RemotingClient application:

```
csc /r:ITest.dll RemotingClient.cs
```

To run the RemotingClient application, ensure the client.config file is in the same directory as the application. Figure 19-6 shows the results of running the RemotingServer and RemotingClient applications having been configured with configuration files.

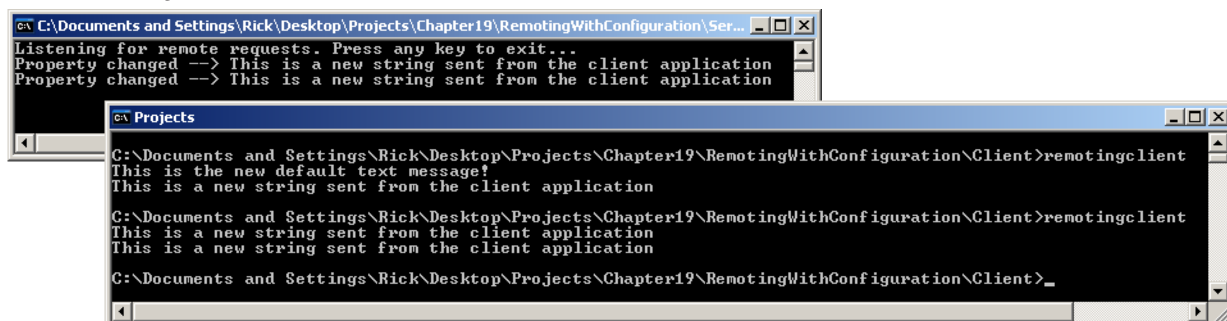


Figure 19-6: Results of Running RemotingServer and RemotingClient with Configuration Files

 © 2008 Rick Miller — All Rights Reserved

Passing Objects Between Client And Server

Remote object method calls can take parameters and return object's just like ordinary objects. A user-defined type intended for transmission across a network must be tagged with the Serializable attribute. Let's see how this is done. The following extended example shows how a remoting client can access a remoting server to get a list of Person objects. To keep things relatively simple, the remote object creates and populates a collection of Person objects. (These Person objects could easily be retrieved from a database, which you'll see done in the following chapter!)

Example 19.12 gives the code for the Person class, which is used in this application.

19.12 Person.cs

```
1    using System;
2
3    [Serializable]
4    public class Person {
5
6      //enumeration
7      public enum Sex {MALE, FEMALE};
8
9      // private instance fields
10     private String   _firstName;
11     private String   _middleName;
12     private String   _lastName;
13     private Sex      _gender;
14     private DateTime _birthday;
15
16
17     //private default constructor
18     private Person(){}
19
20     public Person(String firstName, String middleName, String lastName,
21                   Sex gender, DateTime birthday){
22        FirstName = firstName;
23        MiddleName = middleName;
24        LastName = lastName;
25        Gender = gender;
26        BirthDay = birthday;
27     }
28
29     // public properties
30     public String FirstName {
31       get { return _firstName; }
32       set { _firstName = value; }
33     }
34
35     public String MiddleName {
36       get { return _middleName; }
37       set { _middleName = value; }
38     }
39
40     public String LastName {
41       get { return _lastName; }
42       set { _lastName = value; }
43     }
44
45     public Sex Gender {
46       get { return _gender; }
47       set { _gender = value; }
48     }
49
50     public DateTime BirthDay {
51       get { return _birthday; }
52       set { _birthday = value; }
53     }
54
55      public int Age {
56        get {
57          int years = DateTime.Now.Year - _birthday.Year;
58          int adjustment = 0;
59            if(DateTime.Now.Month < _birthday.Month){
60              adjustment = 1;
61            }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
62                      adjustment = 1;
63                  }
64          return years - adjustment;
65        }
66      }
67
68     public String FullName {
```


© 2008 Rick Miller — All Rights Reserved

```
69        get { return FirstName + " " + MiddleName + " " + LastName; }
70      }
71
72      public String FullNameAndAge {
73        get { return FullName + " " + Age; }
74      }
75
76      public override String ToString(){
77        return FullName + " is a " + Gender + " who is " + Age + " years old.";
78      }
79
80    } // end Person class
```

Referring to Example 19.12 — the Person class has been tagged as being serializable by the addition on line 3 of the Serializable attribute. Compile this class into a dll using the following compiler command:

```
csc /t:library Person.cs
```

Example 19.13 gives the code for the ISurrealistServer interface.

19.13 ISurrealistServer.cs

```
1    using System;
2    using System.Collections.Generic;
3
4    public interface ISurrealistServer {
5      List<Person> GetSurrealists();
6    }
```

Referring to Example 19.13 — the ISurrealistServer interface declares one method named GetSurrealists(), which returns a list of Person objects. Compile this code into a dll by using the following compiler command:

```
csc /t:library /r:Person.dll ISurrealistServer.cs
```

Example 19.14 gives the code for the SurrealistServer class.

19.14 SurrealistServer.cs

```
1    using System;
2    using System.Collections.Generic;
3
4    public class SurrealistServer : MarshalByRefObject, ISurrealistServer {
5
6      private List<Person> surrealists  = null;
7
8      public SurrealistServer(){
9        this.InitializeSurrealists();
10     }
11
12     public List<Person> GetSurrealists(){
13       Console.WriteLine("Request for surrealists received!");
14       return surrealists;
15     }
16
17     private void InitializeSurrealists(){
18       surrealists = new List<Person>();
19       Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
20        Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
21        Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
22        Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
23        Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
24        Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
25                                                    new DateTime(1887, 07, 28));
26
27        surrealists.Add(p1);
28        surrealists.Add(p2);
29        surrealists.Add(p3);
30        surrealists.Add(p4);
31        surrealists.Add(p5);
32        surrealists.Add(p6);
33     }
34
35   }
```

Referring to Example 19.14 — the SurrealistServer class extends MarshalByRefObject and implements the ISurrealistServer interface. It declares a private field named surrealists. It actually creates the list object and populates it with six Person objects in the body of the InitializeSurrealists() method. The GetSurrealists() method simply returns the list object. So, over the network, the entire list of Person objects is returned to the client application when it calls this method.

Next, let's make some changes to the server.config file, as are shown in Example 19.15

19.15 server.config

```
1    <configuration>
2      <system.runtime.remoting>
3        <application>
```


© 2008 Rick Miller — All Rights Reserved

```
4        <service>
5         <wellknown mode="Singleton" type="SurrealistServer, SurrealistServer"
6                                   objectUri="SurrealistServer" />
7        </service>
8        <channels>
9          <channel ref="tcp" port="8080" />
10       </channels>
11     </application>
12   </system.runtime.remoting>
13  </configuration>
```

Referring to Example 19.15 — the changes made to the server.config file reflect the new name of the remote object class and the name of the service by which it can be accessed. These changes appear in lines 5 and 6.

Finally, the code for RemotingServer remains unchanged from the last example, but I repeat it here for continuity in Example 19.16.

19.16 RemotingServer.cs

```
1   using System;
2   using System.Runtime.Remoting;
3   using System.Runtime.Remoting.Channels;
4   using System.Runtime.Remoting.Channels.Tcp;
5
6   public class RemotingServer {
7     public static void Main(){
8      try {
9        RemotingConfiguration.Configure("server.config", false);
10       Console.WriteLine("Listening for remote requests. Press any key to exit...");
11       Console.ReadLine();
12      }catch(Exception e){
13        Console.WriteLine(e);
14      }
15    }
16  }
```

To compile this application, make sure the Person.dll, ISurrealistServer.dll, and SurrealistServer.dll files are in the same directory and use the following compiler command:

```
csc /r:Person.dll;ISurrealistServer.dll;SurrealistServer.dll
RemotingServer.cs
```

When you have finished compiling the RemotingServer application you can start the server. It's time now to write the code for the RemotingClient application. The RemotingClient code is given in Example 19.17.

19.17 RemotingClient.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.Remoting;
4   using System.Runtime.Remoting.Channels;
5   using System.Runtime.Remoting.Channels.Tcp;
6
7   public class RemotingClient {
8     public static void Main(){
9      try {
10       RemotingConfiguration.Configure("client.config", false);
11       WellKnownClientTypeEntry[] client_types = RemotingConfiguration.GetRegisteredWellKnownClientTypes();
12       ISurrealistServer surrealist_server =
13            (ISurrealistServer)Activator.GetObject(typeof(ISurrealistServer), client_types[0].ObjectUrl );
14
15       List<Person> surrealists = surrealist_server.GetSurrealists();
16       foreach(Person p in surrealists){
17         Console.WriteLine(p);
18       }
19      }catch(Exception e){
20        Console.WriteLine(e);
21      }
22    }
23  }
```

Referring to Example 19.17 — the RemotingClient gets its configuration from the client.config file, which is given in the next example. It then gets a reference to an ISurrealistServer object and calls its GetSurrealists() method. It then iterates over the list of Person objects in the body of the `foreach` statement on line 16 and writes each object's ToString() data to the console.

Example 19.18 gives the contents of the client.config file.

19.18 client.config

```
1   <configuration>
2     <system.runtime.remoting>
3       <application>
4         <client>
5           <wellknown type="ISurrealistServer, ISurrealistServer"
```

 © 2008 Rick Miller — All Rights Reserved

```
6                    url="tcp://localhost:8080/SurrealistServer" />
7          </client>
8        </application>
9      </system.runtime.remoting>
10   </configuration>
```

Referring to Example 19.18 — the changes to the client.config file appear on lines 5 and 6 and reflect the name of the remote object type and the service where it can be found.

To compile the RemotingClient application, make copies of the Person.dll and ISurrealistServer.dll, place them in the client code directory, and use the following compiler command:

```
csc /r:Person.dll;ISurrealistServer.dll RemotingClient.cs
```

Figure 19-7 shows the results of running the RemotingServer and RemotingClient applications.

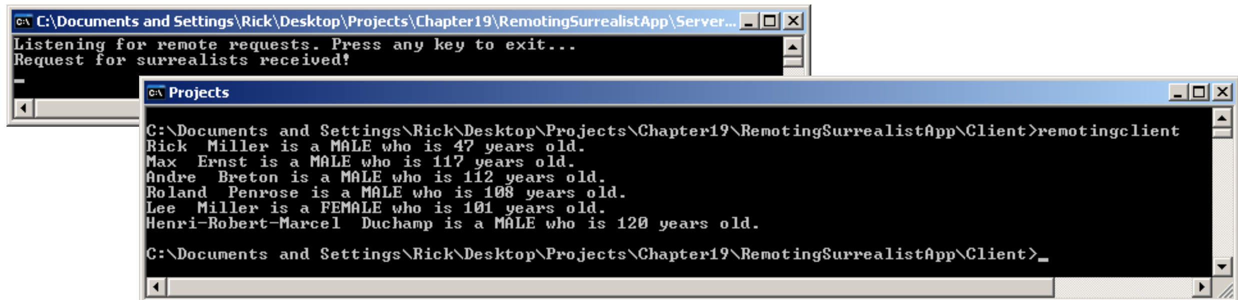


Figure 19-7: Results of Sending a Collection of Person Objects to a Remoting Client

Quick Review

All .NET remoting applications have three common components, regardless of their complexity: a *remotable object*, a *server application* that hosts the remotable object and handles incoming service requests, and a *client application* that utilizes the services of the remotely-hosted object.

A remotable object is created from a class that inherits from System.MarshalByRefObject. This enables the object to be shared across application domains in .NET remoting applications. The remotable object can be simple or complex.

The remoting server application hosts the remotable object and makes its services available via a *channel*. There are three primary channel types: *TcpChannel*, *HttpChannel*, and *IpcChannel*. The IpcChannel is used for *inter-process communication* between client and server applications hosted on the same machine.

The remoting client application accesses the services of the remote object via a *proxy* created automatically by the .NET remoting infrastructure. Once a remoting client application creates a reference to a remote object, it uses the services of the remote object, via the remote object's proxy, as if the remote object were a local object. The underlying complexities associated with calling the remote object's methods or properties are handled automatically by the .NET remoting infrastructure. Remote objects accessed in this manner must extend MarshalByRefObject and any other interfaces as required.

Remote objects can be hosted in *SingleCall* or *Singleton* mode. In SingleCall mode, a new remote object is used to respond to each client service request. In Singleton mode, remote objects persist and maintain state across multiple client service requests.

Remoting client applications can access the services of remote objects via one or more of the remote object's interfaces. This makes changing the implementation of the remote object easier, as long as the new object implements one of the interfaces expected by the client application.

For maximum deployment flexibility, place .NET remoting application deployment data in *configuration files*.

Complex objects sent between remoting client and server applications must be tagged as being serializable by using the *Serializable* attribute.

© 2008 Rick Miller — All Rights Reserved

Client-Server Applications With TcpListener And TcpClient

In this section you'll get a little more down in the weeds with network programming by using the *TcpListener* and *TcpClient* classes to create *client-server applications*. Unlike .NET remoting, you'll need to know how to handle the details of establishing a network connection between client and server applications, how to send data between the client and server so they can perform useful work, and how to use threads to enable a server to handle multiple client requests simultaneously.

TCP/IP Client-Server Overview

The steps required to write a TCP/IP client-server application using the System.Net.TcpListener and System.Net.TcpClient classes are highlighted in the following illustrations.

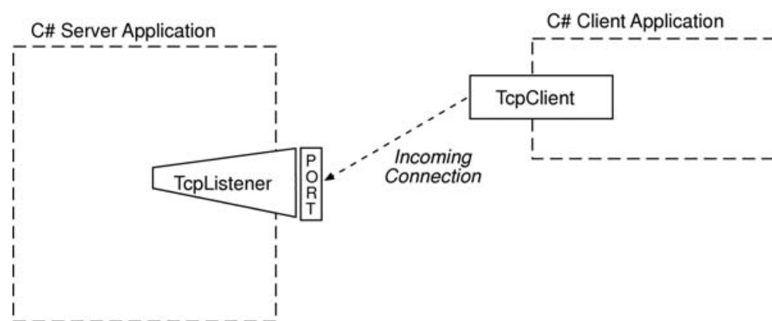


Figure 19-8: Server Application Listens on a Host and Port for Incoming TcpClient Connections

Referring to Figure 19-8 — a server application uses a TcpListener object to listen for incoming TcpClient connections on a particular IP address (or multiple IP addresses) and port number. The client application uses a TcpClient object to connect to a particular machine, given its IP address or DNS name (*i.e.*, www.warrenworks.com) which is then mapped to an IP address, and specified port number.

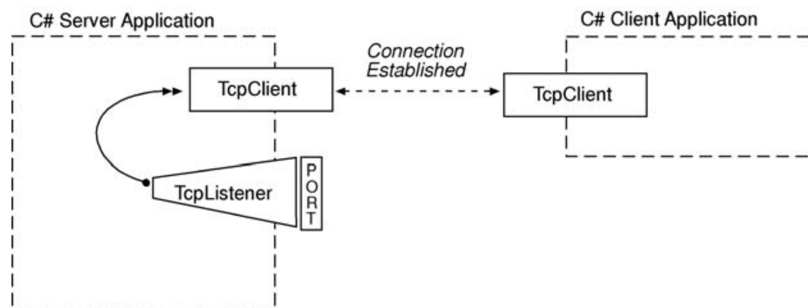


Figure 19-9: TcpListener Accepts Incoming TcpClient Connection

Referring to Figure 19-9 — when the TcpListener object detects an incoming TcpClient connection, it "accepts" the connection, which results in the creation of a server-side TCPclient object. Client-server communication takes place between the server-side and client-side TcpClient objects, as is shown in Figure 19-10.

Both the TcpListener and TcpClient objects provide *wrappers* around *socket* objects. You could use socket objects directly to conduct client-server communication, but doing so is beyond the scope of this book. To learn more about socket programming check out the excellent book *TCP/IP Sockets In C#: Practical Guide for Programers* by David B. Makofske, et. al., ISBN-13: 978-0-12-466051-9.

 © 2008 Rick Miller — All Rights Reserved

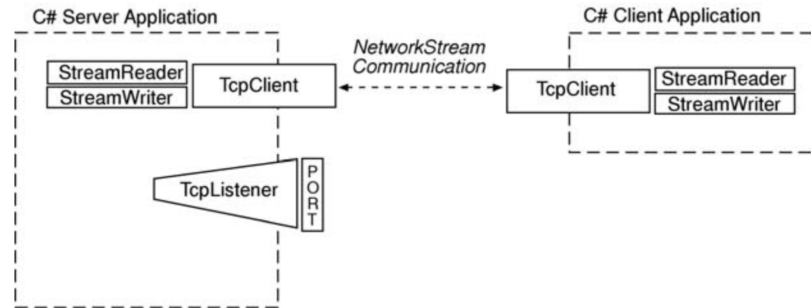


Figure 19-10: TcpClients Communicate via a NetworkStream using StreamReader and StreamWriter Objects

A Simple Client-Server Application

OK, let's put some of what you just learned in the previous section into practical use. The following examples implement a simple client-server application using the TcpListener and TcpClient classes. The application consists of two parts: an EchoServer, which listens for incoming TcpClient connections, and an EchoClient which connects to an EchoServer. When a connection between the EchoServer and EchoClient is established, messages sent from the client to the server are written to the server console and then sent back to the client for display on the client console. Example 19.19 gives the code for the EchoServer application.

19.19 EchoServer.cs

```
1    using System;
2    using System.IO;
3    using System.Net;
4    using System.Net.Sockets;
5
6    public class EchoServer {
7      public static void Main(){
8        TcpListener listener = null;
9        try {
10         listener = new TcpListener(IPAddress.Parse("127.0.0.1"), 8080);
11         listener.Start();
12         Console.WriteLine("EchoServer started...");
13         while(true){
14           Console.WriteLine("Waiting for incoming client connections...");
15           TcpClient client = listener.AcceptTcpClient();
16           Console.WriteLine("Accepted new client connection...");
17           StreamReader reader = new StreamReader(client.GetStream());
18           StreamWriter writer = new StreamWriter(client.GetStream());
19           String s = String.Empty;
20           while(!(s = reader.ReadLine()).Equals("Exit")){
21            Console.WriteLine("From client -> " + s);
22            writer.WriteLine("From server -> " + s);
23            writer.Flush();
24           }
25          reader.Close();
26          writer.Close();
27           client.Close();
28         }
29        }catch(Exception e){
30          Console.WriteLine(e);
31        }finally{
32         if(listener != null){
33           listener.Stop();
34         }
35        }
36     } // end Main()
37   } // end class definition
```

Referring to Example 19.19 — notice first the list of namespaces required for this particular application. It includes System.IO, System.Net, and System.Net.Sockets. The EchoServer application starts by creating an instance of TcpListener, which listens on the local machine IP address of 127.0.0.1 port 8080. (Make sure the port you choose is not in use.) The listener is then started on line 11 by a call to its Start() method. Incoming client connections are processed in the body of the `while` loop, which begins on line 13. On line 15, the listener.AcceptTcpClient() method blocks at that point until it detects an incoming TcpClient connection, at which time it unblocks and returns an instance of TcpClient and assigns it to the client reference. The term *block* refers to a *blocking I/O operation*. The

 © 2008 Rick Miller — All Rights Reserved

EchoServer application effectively stops everything until the AcceptTcpClient() method returns, at which time processing continues.

When the listener detects the incoming TcpClient connection, the application prints a short message stating so to the console, and then, on lines 17 and 18, it creates StreamReader and StreamWriter objects using the client.GetStream() method. The server uses these StreamReader and StreamWriter objects to communicate with the client. On line 19, the application creates a string variable named *s* and uses it to store incoming client strings. The body of the `while` loop, which begins on line 20, processes client-server communication by reading the incoming client string, printing it to the server's console, and then sending it back to the client via the writer.WriteLine() method. Note on line 23 the writer.Flush() method must be called to actually send the string on its way. The `while` loop repeats until the incoming string equals "Exit", at which time the EchoServer returns to listening for new incoming TcpClient connections.

Example 19.20 gives the code for the EchoClient application.

19.20 EchoClient.cs

```
1    using System;
2    using System.IO;
3    using System.Net;
4    using System.Net.Sockets;
5
6    public class EchoClient {
7      public static void Main(){
8       try {
9        TcpClient client = new TcpClient("127.0.0.1", 8080);
10       StreamReader reader = new StreamReader(client.GetStream());
11       StreamWriter writer = new StreamWriter(client.GetStream());
12       String s = String.Empty;
13       while(!s.Equals("Exit")){
14        Console.Write("Enter a string to send to the server: ");
15        s = Console.ReadLine();
16        Console.WriteLine();
17        writer.WriteLine(s);
18        writer.Flush();
19        String server_string = reader.ReadLine();
20        Console.WriteLine(server_string);
21       }
22      reader.Close();
23      writer.Close();
24      client.Close();
25       } catch(Exception e){
26         Console.WriteLine(e);
27       }
28     } // end Main()
29   } // end class definition
```

Referring to Example 19.20 — the EchoClient application creates a TcpClient object that connects to the IP address 127.0.0.1 port 8080. If all goes well, lines 10 and 11 execute and the application creates the StreamReader and StreamWriter objects, which it uses to communicate with the server. On line 12, a string variable named *s* is created and used to send data to the server and to control the processing of the `while` loop, which starts on the following line. On line 15, the Console.ReadLine() method reads a line of text from the console and assigns it to s. On lines 17 and 18, it sends the string s to the server with calls to writer.WriteLine() and writer.Flush(). It then immediately reads the server's response with a call to the reader.ReadLine() method, which assigns the incoming string to the string variable named server_string and then prints the value of server_string to the console. The EchoServer application repeats this processing loop until the user enters the string "Exit" at the console.

To run this application, compile the EchoServer.cs and EchoClient.cs files, start the EchoServer, then run the EchoClient application. Figure 19-11 shows the results of running these applications.

Building A Multithreaded Server

While the previous example served well to illustrate basic client-server principles, the server application in its current form is only able to communicate with one client at a time. In this section, I'll show you how to make a few modifications to EchoServer that will enable it to serve multiple clients simultaneously. A server application that can process multiple simultaneous client connections is referred to as a *multithreaded* server.

The following general steps are required to turn EchoServer into a MultiThreadedEchoServer:

Step 1: Create a separate client processing method that handles network stream communication and other applicable processing between server and client applications.

 © 2008 Rick Miller — All Rights Reserved

Figure 19-11: Results of Running the EchoClient and EchoServer Applications

Step 2: For each incoming client connection, spawn a separate thread, passing to it the name of the client processing method.

That's it! Let's see how these modifications look in the code. Example 19.21 gives the code for the MultiThreadedEchoServer class.

19.21 MultiThreadedEchoServer.cs

```
1    using System;
2    using System.IO;
3    using System.Net;
4    using System.Net.Sockets;
5    using System.Threading;
6
7    public class MultiThreadedEchoServer {
8
9      private static void ProcessClientRequests(Object argument){
10       TcpClient client = (TcpClient)argument;
11       try {
12         StreamReader reader = new StreamReader(client.GetStream());
13         StreamWriter writer = new StreamWriter(client.GetStream());
14         String s = String.Empty;
15         while(!(s = reader.ReadLine()).Equals("Exit")){
16           Console.WriteLine("From client -> " + s);
17           writer.WriteLine("From server -> " + s);
18           writer.Flush();
19         }
20         reader.Close();
21         writer.Close();
22         client.Close();
23         Console.WriteLine("Closing client connection!");
24       }catch(IOException){
25         Console.WriteLine("Problem with client communication. Exiting thread.");
26       }finally{
27          if(client != null){
28            client.Close();
29          }
30        }
31     }
32
33     public static void Main(){
34       TcpListener listener = null;
35       try {
36         listener = new TcpListener(IPAddress.Parse("127.0.0.1"), 8080);
37         listener.Start();
38         Console.WriteLine("MultiThreadedEchoServer started...");
39         while(true){
40           Console.WriteLine("Waiting for incoming client connections...");
41           TcpClient client = listener.AcceptTcpClient();
42           Console.WriteLine("Accepted new client connection...");
43           Thread t = new Thread(ProcessClientRequests);
44           t.Start(client);
45         }
46        }catch(Exception e){
47          Console.WriteLine(e);
48        }finally{
49          if(listener != null){
50            listener.Stop();
51          }
```


© 2008 Rick Miller — All Rights Reserved

```
52          }
53      } // end Main()
54    } // end class definition
```

Referring to Example 19.21 — first, a new namespace, System.Threading, has been added to list of using directives to gain access to the Thread class. I created a new method on line 9 named ProcessClientRequests(). Note that this method takes one argument of type Object, which is cast immediately to a TcpClient object. The reason this cast is necessary is because the ProcessClientRequests() method has the signature of a ParameterizedThreadStart delegate, which specifies one argument of type Object. You'll see how this method is actually used in the body of the Main() method.

I copied the bulk of the ProcessClientRequests() method from the previous version of EchoClient starting with the creation of the StreamReader and StreamWriter objects. It includes the whole of the second, or inner, `while` loop. I enclosed the method's code within its own `try/catch/finally` block because once the separate thread begins execution, it must handle any exceptions it generates.

In the body of the Main() method, the TcpListener object is created as before on line 36 and is started on line 37. The `while` loop beginning on line 39 repeats forever waiting for incoming client connections. When it detects an incoming client connection, the AcceptTcpClient() method returns a reference to a new TcpClient object and processing continues with the creation of a new Thread object on line 43. The name of the method this thread will execute, ProcessClientRequests, is passed to the Thread constructor. An alternative call to the Thread constructor could look like this:

```
Thread t = new Thread(new ParameterizedThreadStart(ProcessClientRequests));
```

In this example, the ParameterizedThreadStart delegate object is explicitly created and passed to the Thread constructor. The new thread is started with a call to t.Start() on line 44, passing to it the reference to the TcpClient object named *client*. When line 44 completes execution, the `while` loop continues and the server returns to listening for incoming client connections.

You now have a multithreaded server application! The code for EchoClient, given in the previous section, remains unchanged.

Figure 19-12 shows the results of running the MultiThreadedServer and connecting to it from two EchoClient applications.

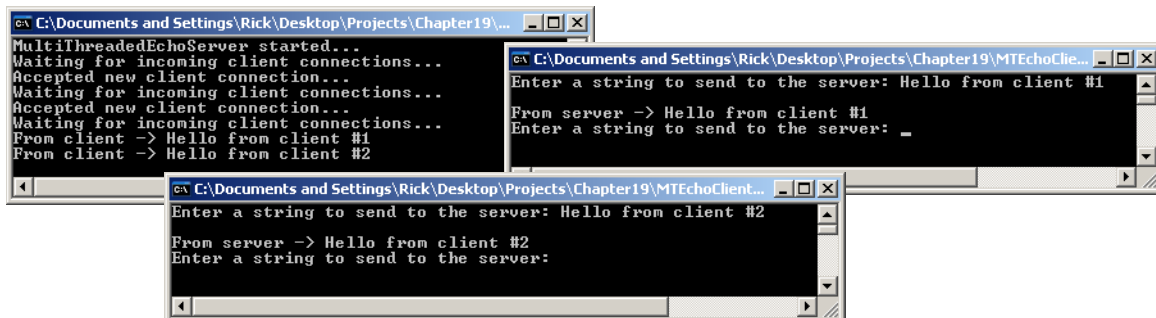


Figure 19-12: Two Clients Connected to MultiThreadedClientServer

Listening On Multiple IP Addresses

The MultiThreadedServer has some nice functionality, but in its current form the TcpListener binds to only one server IP address, which in the previous examples has been the local loopback adapter 127.0.0.1. It would be nice to listen on several IP addresses simultaneously. The only change necessary to do this is to use "IPAddress.Any" when creating the TcpListener object. Example 19.22 gives the code for the MultiIPEchoServer class.

19.22 MultiIPEchoServer.cs

```
1     using System;
2     using System.Drawing;
3     using System.IO;
4     using System.Net;
5     using System.Net.Sockets;
6     using System.Net.NetworkInformation;
7     using System.Threading;
8
9     public class MultiIPEchoServer {
10
```

 © 2008 Rick Miller — All Rights Reserved

```
11     private static void ProcessClientRequests(Object argument){
12       TcpClient client = (TcpClient)argument;
13       try {
14         StreamReader reader = new StreamReader(client.GetStream());
15         StreamWriter writer = new StreamWriter(client.GetStream());
16         String s = String.Empty;
17         while(!(s = reader.ReadLine()).Equals("Exit")){
18           Console.WriteLine("From client -> " + s);
19           writer.WriteLine("From server -> " + s);
20           writer.Flush();
21         }
22         reader.Close();
23         writer.Close();
24         client.Close();
25         Console.WriteLine("Client connection closed!");
26       }catch(IOException){
27         Console.WriteLine("Problem with client communication. Exiting thread.");
28       }finally{
29          if(client != null){
30            client.Close();
31          }
32        }
33     }
34
35     private static void ShowServerNetworkConfig(){
36       Console.ForegroundColor = ConsoleColor.Yellow;
37       NetworkInterface[] adapters = NetworkInterface.GetAllNetworkInterfaces();
38       foreach(NetworkInterface adapter in adapters){
39         Console.WriteLine(adapter.Description);
40         Console.WriteLine("\tAdapter Name: " + adapter.Name);
41         Console.WriteLine("\tMAC Address: " + adapter.GetPhysicalAddress());
42         IPInterfaceProperties ip_properties = adapter.GetIPProperties();
43         UnicastIPAddressInformationCollection addresses = ip_properties.UnicastAddresses;
44         foreach(UnicastIPAddressInformation address in addresses){
45           Console.WriteLine("\tIP Address: " + address.Address);
46         }
47       }
48       Console.ForegroundColor = ConsoleColor.White;
49     }
50
51     public static void Main(){
52       TcpListener listener = null;
53       try {
54         ShowServerNetworkConfig();
55         listener = new TcpListener(IPAddress.Any, 8080);
56         listener.Start();
57         Console.WriteLine("MultiIPEchoServer started...");
58         while(true){
59           Console.WriteLine("Waiting for incoming client connections...");
60           TcpClient client = listener.AcceptTcpClient();
61           Console.WriteLine("Accepted new client connection...");
62           Thread t = new Thread(ProcessClientRequests);
63           t.Start(client);
64         }
65        }catch(Exception e){
66          Console.WriteLine(e);
67        }finally{
68          if(listener != null){
69            listener.Stop();
70          }
71         }
72     } // end Main()
73   } // end class definition
```

Referring to Example 19.22 — I have added another method to the server code named ShowServerNetworkConfig() which begins on line 35. I've also added another namespace, System.Net.NetworkInformation, to the list of `using` directives.

Referring to the ShowServerNetworkConfig() method — the first thing it does is set Console.ForegroundColor to Color.Yellow. This makes the network information stand out from the ordinary client-server interaction messages. Next, on line 37, the NetworkInterface.GetAllNetworkInterfaces() method is called. This returns an array of NetworkInterface objects. The `foreach` statement starting on line 38 iterates over the array of NetworkInterface objects and prints out various properties about each one including the interface's *Description*, *Name*, and *Physical* or *MAC* addresses. On line 42, I create an IPInterfaceProperties object with the help of the adapter.GetIPProperties() method and use it to get a collection of UnicastIPAddressInformation objects for each adapter. The `foreach` loop starting on line 44 iterates over the collection of UnicastIPAddressInformation objects and prints each IP address to the console. Finally, the method concludes by resetting the Console.ForegroundColor to Color.White.

© 2008 Rick Miller — All Rights Reserved

The only changes to the Main() method include the addition of line 54, where I make a call to the ShowServerNetworkConfig() method, and on line 55 where I bind the TcpListener object to all available machine IP addresses by using IPAddress.Any.

The EchoClient has been changed to connect to an IP address given in the form of a command-line argument when the program executes. The code for the modified EchoClient class is given in Example 19.23.

19.23 EchoClient.cs (Mod 1)

```
1    using System;
2    using System.IO;
3    using System.Net;
4    using System.Net.Sockets;
5
6    public class EchoClient {
7      public static void Main(String[] args){
8       IPAddress ip_address = IPAddress.Parse("127.0.0.1"); //default
9       int port = 8080;
10      try{
11        if(args.Length >= 1){
12          ip_address = IPAddress.Parse(args[0]);
13        }
14      }catch(FormatException){
15        Console.WriteLine("Invalid IP address entered. Using default IP of: " + ip_address.ToString());
16      }
17      try {
18       Console.WriteLine("Attempting to connect to server at IP address: {0} port: {1}",
19                                                     ip_address.ToString(), port);
20       TcpClient client = new TcpClient(ip_address.ToString(), port);
21       Console.WriteLine("Connection successful!");
22       StreamReader reader = new StreamReader(client.GetStream());
23       StreamWriter writer = new StreamWriter(client.GetStream());
24       String s = String.Empty;
25       while(!s.Equals("Exit")){
26        Console.Write("Enter a string to send to the server: ");
27        s = Console.ReadLine();
28        Console.WriteLine();
29        writer.WriteLine(s);
30        writer.Flush();
31        if(!s.Equals("Exit")){
32          String server_string = reader.ReadLine();
33          Console.WriteLine(server_string);
34        }
35       }
36       reader.Close();
37       writer.Close();
38       client.Close();
39       }catch(Exception e){
40         Console.WriteLine(e);
41       }
42     } // end Main()
43    } // end class definition
```

Referring to Example 19.23 — the EchoClient class now checks the argument array for the presence of a valid IP address. If the given IP address is malformed, the IPAddress.Parse() method throws an exception and assigns the default IP address value of 127.0.0.1 to the ip_address field. On line 20, the ip_address field is used in the TcpClient constructor call where it is converted into a string. The remainder of the code remains unchanged from the previous example.

To run these versions of the client and server applications, compile the code and start the MultiIPEchoServer, then run the modified EchoClient application. The results of these changes can be seen in Figure 19-13. Note how each client connects to the server via a different IP address.

Sending Objects Between Client And Server

In the previous examples, I've limited the exchange between client and server application to strings. In this section I'll show you how to serialize a complex object on the server side and send it to the client for deserialization. Remember that in the case of .NET remoting applications, the hard work of serializing complex objects is done for you by the remoting framework. Not here, no, no, no. If you want to serialize a complex object and send it across the network you'll need to get your hands dirty.

The following two examples implement a SurrealistEchoServer. The application actually consists of three classes: SurrealistEchoServer.cs, SurrealistDB.cs, and Person.cs, which is not repeated here. Example 19.24 gives the code for the SurrealistEchoServer class.

 © 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\MultiIPMTClientServer\Ser...
Microsoft Loopback Adapter
        Adapter Name: Local Area Connection 2
        MAC Address: 02004C4F4F50
        IP Address: 10.10.10.10
Broadcom NetXtreme 57xx Gigabit Controller - Packet Scheduler Miniport
        Adapter Name: Local Area Connection
        MAC Address: 0011114C00E3
        IP Address: 192.168.1.104
VMware Virtual Ethernet Adapter for VMnet1
        Adapter Name: VMware Network Adapter VMnet1
        MAC Address: 005056C00001
        IP Address: 192.168.186.1
VMware Virtual Ethernet Adapter for VMnet8
        Adapter Name: VMware Network Adapter VMnet8
        MAC Address: 005056C00008
        IP Address: 192.168.121.1
MS TCP Loopback interface
        Adapter Name: MS TCP Loopback interface
        MAC Address:
        IP Address: 127.0.0.1
MultiIPEchoServer started...
Waiting for incoming client connections...
Accepted new client connection...
Waiting for incoming client connections...
Accepted new client connection...
Waiting for incoming client connections...
From client -> Hello from client #1
From client -> Hello from client #2
```

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\MultiIPMTClientServer\Client...
Attempting to connect to server at IP address: 192.168.1.104 port: 8080
Connection successful!
Enter a string to send to the server: Hello from client #1

From server -> Hello from client #1
Enter a string to send to the server: _
```

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\MultiIPMTClientServer\Client\...
Attempting to connect to server at IP address: 192.168.121.1 port: 8080
Connection successful!
Enter a string to send to the server: Hello from client #2

From server -> Hello from client #2
Enter a string to send to the server: _
```

Figure 19-13: Results of Running MultiIPEchoServer and EchoClient (Mod 1) Applications

19.24 SurrealistEchoServer.cs

```
1     using System;
2     using System.Drawing;
3     using System.IO;
4     using System.Net;
5     using System.Net.Sockets;
6     using System.Net.NetworkInformation;
7     using System.Threading;
8     using System.Runtime.Serialization;
9     using System.Runtime.Serialization.Formatters.Binary;
10
11    public class SurrealistEchoServer {
12
13      private static void ProcessClientRequests(Object argument){
14        TcpClient client = (TcpClient)argument;
15        try {
16          StreamReader reader = new StreamReader(client.GetStream());
17          StreamWriter writer = new StreamWriter(client.GetStream());
18          String s = String.Empty;
19          while(!(s = reader.ReadLine()).Equals("Exit")){
20            switch(s){
21              case "GetSurrealists" : {
22                Console.WriteLine("From client -> " + s);
23                SerializeSurrealists(client.GetStream());
24                client.GetStream().Flush();
25                break;
26              }
27              default: {
28                Console.WriteLine("From client -> " + s);
29                writer.WriteLine("From server -> " + s);
30                writer.Flush();
31                break;
32              }
33            } // end switch
34          } // end while
35          reader.Close();
```


© 2008 Rick Miller — All Rights Reserved

```
36          writer.Close();
37          client.Close();
38          Console.WriteLine("Client connection closed!");
39        }catch(IOException){
40          Console.WriteLine("Problem with client communication. Exiting thread.");
41        }catch(NullReferenceException){
42          Console.WriteLine("Incoming string was null! Client may have terminated prematurly.");
43        }catch(Exception e){
44          Console.WriteLine("Unknown exception occured.");
45          Console.WriteLine(e);
46        }finally{
47           if(client != null){
48             client.Close();
49           }
50         }
51      } // end ProcessClientRequests()
52
53      private static void SerializeSurrealists(NetworkStream stream){
54        SurrealistDB db = new SurrealistDB();
55        BinaryFormatter bf = new BinaryFormatter();
56        bf.Serialize(stream, db.GetSurrealists());
57      } // end SerializeSurrealists()
58
59      private static void ShowServerNetworkConfig(){
60        Console.ForegroundColor = ConsoleColor.Yellow;
61        NetworkInterface[] adapters = NetworkInterface.GetAllNetworkInterfaces();
62        foreach(NetworkInterface adapter in adapters){
63          Console.WriteLine(adapter.Description);
64          Console.WriteLine("\tAdapter Name: " + adapter.Name);
65          Console.WriteLine("\tMAC Address: " + adapter.GetPhysicalAddress());
66          IPInterfaceProperties ip_properties = adapter.GetIPProperties();
67          UnicastIPAddressInformationCollection addresses = ip_properties.UnicastAddresses;
68          foreach(UnicastIPAddressInformation address in addresses){
69            Console.WriteLine("\tIP Address: " + address.Address);
70          }
71        }
72        Console.ForegroundColor = ConsoleColor.White;
73      } // end ShowServerNetworkConfig()
74
75      public static void Main(){
76        TcpListener listener = null;
77        try {
78          ShowServerNetworkConfig();
79          listener = new TcpListener(IPAddress.Any, 8080);
80          listener.Start();
81          Console.WriteLine("SurrealistEchoServer started...");
82          while(true){
83            Console.WriteLine("Waiting for incoming client connections...");
84            TcpClient client = listener.AcceptTcpClient();
85            Console.WriteLine("Accepted new client connection...");
86            Thread t = new Thread(ProcessClientRequests);
87            t.Start(client);
88          }
89         }catch(Exception e){
90           Console.WriteLine(e);
91         }finally{
92           if(listener != null){
93             listener.Stop();
94           }
95          }
96      } // end Main()
97    } // end class definition
```

Referring to Example 19.24 — first, note the addition of several namespaces required to perform object serialization. These include System.Runtime.Serialization and System.Runtime.Serialization.Formatters.Binary.

The SurrealistEchoServer class's ProcessClientRequests() method has been slightly modified. It echoes client strings as before, but if the client string equals “GetSurrealists”, it returns to the client a serialized collection of Person objects. It does this with a call to its SerializeSurrealists() method, which begins on line 53.

The SerializeSurrealists() method takes a NetworkStream object as an argument. On line 54, it creates an instance of SurrealistsDB followed by the creation of a BinaryFormatter object on the next line. The BinaryFormatter serializes the List<Person> object returned by the db.GetSurrealists() method into the stream. When the SerializeSurrealists() method returns, the network stream is flushed to send the collection of People objects on their way to the client.

Example 19.25 gives the code for the SurrealistDB class.

 © 2008 Rick Miller — All Rights Reserved

19.25 SurrealistDB.cs

```
1    using System;
2    using System.Collections.Generic;
3
4    public class SurrealistDB {
5
6      private List<Person> surrealists = null;
7
8      public SurrealistDB(){
9        this.InitializeSurrealists();
10     }
11
12     public List<Person> GetSurrealists(){
13       return surrealists;
14     }
15
16     private void InitializeSurrealists(){
17       surrealists = new List<Person>();
18       Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
19        Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
20        Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
21        Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
22        Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
23        Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
24                                                 new DateTime(1887, 07, 28));
25
26        surrealists.Add(p1);
27        surrealists.Add(p2);
28        surrealists.Add(p3);
29        surrealists.Add(p4);
30        surrealists.Add(p5);
31        surrealists.Add(p6);
32     }
33   } // end class definition
```

Referring to Example 19.25 — The SurrealistDB class initializes a list of People objects and provides a GetSurrealists() method which returns the populated list. (**Note:** This class could easily have been written to connect to a data base to fetch the required information. You'll see how that's done in Chapter 20.)

The EchoClient class must be modified to accept and deserialize the incoming list of People objects. Example 19.26 gives the code for the modified EchoClient class.

19.26 EchoClient.cs (Mod 2)

```
1    using System;
2    using System.IO;
3    using System.Net;
4    using System.Net.Sockets;
5    using System.Runtime.Serialization;
6    using System.Runtime.Serialization.Formatters.Binary;
7    using System.Collections.Generic;
8
9    public class EchoClient {
10
11
12     static List<Person> DeserializeSurrealists(NetworkStream stream){
13       BinaryFormatter bf = new BinaryFormatter();
14       return (List<Person>)bf.Deserialize(stream);
15     }
16
17     static void WriteSurrealistDataToConsole(List<Person> surrealists){
18       foreach(Person p in surrealists){
19         Console.WriteLine(p);
20       }
21     }
22
23     public static void Main(String[] args){
24      IPAddress ip_address = IPAddress.Parse("127.0.0.1"); //default
25      int port = 8080;
26      try{
27        if(args.Length >= 1){
28          ip_address = IPAddress.Parse(args[0]);
29        }
30      }catch(FormatException){
31        Console.WriteLine("Invalid IP address entered. Using default IP of: " + ip_address.ToString());
32      }
33      try {
34       Console.WriteLine("Attempting to connect to server at IP address: {0} port: {1}",
35                                                   ip_address.ToString(), port);
36       TcpClient client = new TcpClient(ip_address.ToString(), port);
37       Console.WriteLine("Connection successful!");
38       StreamReader reader = new StreamReader(client.GetStream());
```


© 2008 Rick Miller — All Rights Reserved

```
39        StreamWriter writer = new StreamWriter(client.GetStream());
40        String s = String.Empty;
41        while(!s.Equals("Exit")){
42         Console.Write("Enter \"GetSurrealists\" to retrieve list from server: ");
43         s = Console.ReadLine();
44         Console.WriteLine();
45         switch(s){
46           case "GetSurrealists" : {
47             writer.WriteLine(s);
48             writer.Flush();
49             WriteSurrealistDataToConsole(DeserializeSurrealists(client.GetStream()));
50             Console.WriteLine();
51             break;
52           }
53           case "Exit" : {
54             writer.WriteLine(s);
55             writer.Flush();
56             break;
57           }
58           default: {
59             writer.WriteLine(s);
60             writer.Flush();
61             String server_string = reader.ReadLine();
62             Console.WriteLine(server_string);
63             Console.WriteLine();
64             break;
65           }
66         }
67        }
68        reader.Close();
69        writer.Close();
70        client.Close();
71        }catch(Exception e){
72          Console.WriteLine(e);
73        }
74      } // end Main()
75    } // end class definition
```

Referring to Example 19.26 — the modified EchoClient application sends strings to the server as before. When the string it sends equals "GetSurrealists" the server returns a serialized list of People objects. *(i.e.*, List<People>) The client must then deserialize the object and cast it to its expected type, which it does with the DeserializeSurrealists() method. Once the list of People objects is deserialized, the EchoClient application calls the WriteSurrealistDataToConsole() method. All this action takes place on line 49!

To run these applications, copy the Person.dll into both client and server directories, then change to the server directory and compile the server application using the following compiler commands.

First compile the SurrealistDB class into a dll:

```
csc /t:library /r:Person.dll SurrealistDB.cs
```

Then compile the server itself:

```
csc /r:Person.dll;SurrealistDB.dll SurrealistEchoServer.cs
```

Change to the client directory and compile the EchoClient class like so:

```
csc /r:Person.dll EchoClient.cs
```

Finally, start the SurrealistEchoServer application and then run the EchoClient application. Figure 19-14 gives the results of fetching some surrealists from the server.

Quick Review

When building client-server applications using the *TcpListener* and *TcpClient* classes, you'll need to know how to handle the details of establishing the network connection between client and server applications, how to send data between the client and server so they can perform useful work, and how to use threads to enable a server to handle multiple client requests simultaneously.

The TcpListener and TcpClient classes provide *wrappers* around *Socket* objects. You can access Socket objects directly if you need more control over client-server network communication.

The general steps required to create a client-server application using TcpListener and TcpClient include the following: 1) create a TcpListener object that listens for incoming TcpClient connections on a specified IP address and port number, 2) on the client side, create a TcpClient object that connects to a particular server identified by an IP address and a particular port number, 3) when the listener detects an incoming TcpClient connection its AcceptTcpClient() method returns (unblocks) and creates a server-side TcpClient object, 4) use the TcpClient's GetStream()

 © 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\SurrealistClientServer\Ser...
Microsoft Loopback Adapter
        Adapter Name: Local Area Connection 2
        MAC Address: 02004C4F4F50
        IP Address: 10.10.10.10
Broadcom NetXtreme 57xx Gigabit Controller - Packet Scheduler Miniport
        Adapter Name: Local Area Connection
        MAC Address: 0011114C00E3
        IP Address: 192.168.1.104
VMware Virtual Ethernet Adapter for VMnet1
        Adapter Name: VMware Network Adapter VMnet1
        MAC Address: 005056C00001
        IP Address: 192.168.186.1
VMware Virtual Ethernet Adapter for VMnet8
        Adapter Name: VMware Network Adapter VMnet8
        MAC Address: 005056C00008
        IP Address: 192.168.121.1
MS TCP Loopback interface
        Adapter Name: MS TCP Loopback interface
        MAC Address:
        IP Address: 127.0.0.1
SurrealistEchoServer started...
Waiting for incoming client connections...
Accepted new client connection...
Waiting for incoming client connections...
From client -> Hello!
From client -> GetSurrealists
_
```

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter19\SurrealistClientServer\Client...
Attempting to connect to server at IP address: 127.0.0.1 port: 8080
Connection successful!
Enter "GetSurrealists" to retrieve list from server: Hello!

From server -> Hello!

Enter "GetSurrealists" to retrieve list from server: GetSurrealists

Rick  Miller is a MALE who is 47 years old.
Max  Ernst is a MALE who is 117 years old.
Andre  Breton is a MALE who is 112 years old.
Roland  Penrose is a MALE who is 108 years old.
Lee  Miller is a FEMALE who is 101 years old.
Henri-Robert-Marcel  Duchamp is a MALE who is 121 years old.

Enter "GetSurrealists" to retrieve list from server: _
```

Figure 19-14: Results of Running SurrealistEchoServer and EchoClient (Mod 2)

method to get a reference to the NetworkStream object, 5) use the NetworkStream object to create StreamReader and StreamWriter objects. Remember to follow a call to StreamWriter.Write() with a call to StreamWriter.Flush().

Multithreaded servers can service multiple simultaneous client connections. Use the Thread class to create a separate client processing thread. This frees up the server to listen for new incoming client connections.

Objects passed between client-server applications must be tagged Serializable. Use a BinaryFormatter to serialize an object to the NetworkStream. Call NetworkStream.Flush() to send the object on its way.

Summary

All .NET remoting applications have three common components, regardless of their complexity: a *remotable object*, a *server application* that hosts the remotable object and handles incoming service requests, and a *client application* that utilizes the services of the remotely-hosted object.

A remotable object is created from a class that inherits from System.MarshalByRefObject. This enables the object to be shared across application domains in .NET remoting applications. Note that the remotable object can be simple or complex.

The remoting server application hosts the remotable object and makes its services available via a *channel*. There are three primary channel types: *TcpChannel*, *HttpChannel*, and *IpcChannel*. The IpcChannel is used for *inter-process* communication between client and server applications hosted on the same machine.

The remoting client application accesses the services of the remote object via a *proxy* created automatically by the .NET remoting infrastructure. Once a remoting client application creates a reference to a remote object, it uses the services of the remote object, via the remote object's proxy, as if the remote object were a local object. The underlying complexities associated with calling the remote object's methods or properties are handled automatically by the

© 2008 Rick Miller — All Rights Reserved

.NET remoting infrastructure. Remote objects accessed in this manner must extend MarshalByRefObject and any other interfaces as required.

Remote objects can be hosted in *SingleCall* or *Singleton* mode. In SingleCall mode, a new remote object is used to respond to each client service request. In Singleton mode, remote objects persist and maintain state across multiple client service requests.

Remoting client applications can access the services of remote objects via one or more of the remote object's interfaces. This makes changing the implementation of the remote object easier, as long as the new object implements one of the interfaces expected by the client application.

For maximum deployment flexibility, place .NET remoting application deployment data in *configuration files*.

Complex objects sent between remoting client and server applications must be tagged as being serializable by using the *Serializable* attribute.

When building client-server applications using the *TcpListener* and *TcpClient* classes, you'll need to know how to handle the details of establishing the network connection between client and server applications, how to send data between the client and server so they can perform useful work, and how to use threads to enable a server to handle multiple client requests simultaneously.

The TcpListener and TcpClient classes provide *wrappers* around *Socket* objects. You can access Socket objects directly if you need more control over client-server network communication.

The general steps required to create a client-server application using TcpListener and TcpClient include the following: 1) create a TcpListener object that listens for incoming TcpClient connections on a specified IP address and port number, 2) on the client side, create a TcpClient object that connects to a particular server identified by an IP address and a particular port number, 3) when the listener detects an incoming TcpClient connection, its AcceptTcpClient() method returns (unblocks) and creates a server-side TcpClient object, 4) use the TcpClient's GetStream() method to get a reference to the NetworkStream object, 5) use the NetworkStream object to create StreamReader and StreamWriter objects. Remember to follow a call to StreamWriter.Write() with a call to StreamWriter.Flush().

Multithreaded servers can service multiple simultaneous client connections. Use the Thread class to create a separate client processing thread. This frees up the server to listen for new incoming client connections.

Objects passed between client-server applications must be tagged *Serializable*. Use a BinaryFormatter to serialize an object to the NetworkStream. Call NetworkStream.Flush() to send the object on its way.

Skill-Building Exercises

1. **API Drill:** Visit the System.Runtime.Remoting, System.Runtime.Remoting.Channels, and System.Runtime.Remoting.Channels.Tcp namespaces and list each class, structure, interface, and enumeration. Write a brief description of its purpose. Browse each entry's members including its methods and properties.

2. **Programming Drill:** Compile and execute all the exercises in this chapter.

3. **Code Drill:** Trace the execution of all the exercises in this chapter.

4. **Programming Drill:** Modify this chapter's .NET remoting examples to use the HttpChannel.

5. **API Drill:** Explore the System.Net and System.Net.Sockets namespaces and list each class, structure, interface, and enumeration. Write a brief description of its purpose. Browse each entry's members including its methods and properties.

6. **Programming Drill:** Modify this chapter's client-server examples to use UDP vs. TCP.

7. **Programming Drill:** Modify this chapter's client-server examples to better handle the possibility of a network outage between client and server applications. For example, modify the EchoClient to gracefully recover if it tries to send something to a server but experiences a long network delay. (**Hint:** Explore the TcpClient class's properties section on MSDN.)

© 2008 Rick Miller — All Rights Reserved

8. **Extra Reading:** Procure the book *TCP/IP SOCKETS IN C#: Practical Guide for Programmers* and read it from front to back!

9. **Programming Drill:** Is a .NET remoting server multithreading capable? To answer this question, modify the last RemotingClient example given in Example 19.17 so that it repeatedly sets the Text property on the remote object and then goes to sleep (*i.e.*, Thread.Sleep()) for 3 seconds. Start the RemotingServer application and then start two or more RemotingClient applications and see what happens.

10. **Deployment Drill:** Deploy and test this chapter's example applications on different machines. That is, start the server on one machine and run the client application on another machine on the same network. (**Note:** Only the client applications capable of connecting to IP addresses other than 127.0.0.1 will work in this scenario.)

Suggested Projects

1. **Network Robot Rat:** Write a client-server version of the robot rat application. Create a server application that displays robot rat images in a GUI representation of the floor. Each incoming client connection should have their very own image of robot rat displayed and moved on the floor. Use the client application to control the movements of the robot rat. In the client application show an image of the floor, which gives the position of that client's robot rat. Alternatively, give the client application a spy capability (enabled by the server) that lets it see the positions of all the other connected clients's robot rats.

2. **Network Employee Management Application:** Write a client-server application that lets users remotely access and manipulate a file containing employee information. Use the client application to view a list of employees, add a new employee, edit an existing employee, and create new employees. Use the Employee example code given in Chapter 11. Give the client application a graphical user interface.

3. **Chat Program:** Write a program that lets multiple users connect and chat. The server should request the user name from new client connections. The client application should be able to see a list of connected users. Have the server echo user messages to all connected clients.

4. **Email Client:** Study the members of the System.Net.Mail namespace. Write a client program that lets you connect to your email provider, download and read your messages, and create and send new messages.

Self-Test Questions

1. What three things do all .NET remoting applications have in common?

2. What class must be extended to create a remotable object?

3. What attribute must you tag a class with before you can transmit objects of its type between .NET remoting applications?

4. List and briefly describe the purpose of the three primary remoting channel types.

5. What is the primary benefit derived from accessing a remote object via an interface?

6. What's the difference between a remote object deployed in the *SingleCall* mode vs. the *Singleton* mode?

7. Describe the roles of the TcpListener and TcpClient classes in a typical client-server application.

© 2008 Rick Miller — All Rights Reserved

8. What happens when you call the TcpListener.AcceptTcpClient() method?

9. Describe in general terms what you need to do to create a multithreaded server application.

10. Why must you follow a call to NetworkStream.Write() with a call to NetworkStream.Flush()?

References

David B. Makofske, et. al. *TCP/IP SOCKETS IN C#: Practical Guide for Programmers*. Morgan Kaufmann Publishers, 2004, ISBN-13: 978-0-12-466051-9, ISBN-10:0-12-466051-7

Microsoft Developer Network (MSDN) *.NET Framework 3.0 and3.5 Reference Documentation* [www.msdn.com]

Notes

© 2008 Rick Miller — All Rights Reserved

Chapter 20

Pentax 67 / SMC Takumar 55/2.8 / Kodak Tri-X Professional

WCC Interior

Database Access & Multitiered Applications

Learning Objectives

- Design and build a multitiered, networked, data-driven, client-server application
- Use Structured Query Language (SQL) to manipulate a relational database
- State the definition of the terms "table", "row", "column", "primary key", "foreign key", and "constraint"
- Use data access objects (DAOs) to map objects to relational database tables
- Use business objects (BOs) to implement business logic
- Use value objects (VOs) to model application entities
- Use Microsoft Enterprise Library Data Access Block to build a data-driven, client-server application
- Use the DatabaseFactory class to create a database connection
- Use prepared statements to execute SQL commands
- Use prepared statement parameters to build dynamic SQL commands
- Correlate a C# data type to its corresponding Microsoft SQL Server data type
- Manipulate large binary database objects
- Use a DataGridView to display and manipulate tabular data in a graphical user interface (GUI)

© 2008 Rick Miller — All Rights Reserved

Introduction

As you might have guessed from reading the learning objectives, we have a lot to talk about in this chapter. The relational database topic is large enough on its own to warrant a complete book, and many excellent texts have already been written, so I will limit my discussion about this topic to only the essentials you need to know to get up to speed quick.

I will also take a different approach in my presentation of ADO.NET. It's too feature rich to cover completely in great detail, so I am omitting broad swaths of it to concentrate on those aspects I feel give you more power and flexibility to design extremely complex database applications. Instead of DataSets and DataProviders I will show you how to create and use Data Access Objects (DAOs), Business Objects (BOs), and Value Objects (VOs).

Your success in completing this chapter hinges on your ability to properly install and configure several critical components. These include Microsoft SQLServer Express Edition and Microsoft Enterprise Library Application Blocks. The installation of SQLServer Express is relatively painless and straightforward. The installation of the Enterprise Library Application Blocks will seem daunting to novice programmers, especially if you're not familiar with using the command-line console. (If you've made it this far in the book, you should be getting pretty good at using it by now!)

Note: You may have to fiddle with things to get them to work! As you know by now, programming, in large part, is a constant attention-to-detail drill. At no other time is this more true than when you start adding the complexities of database access to the mix. One small spelling mistake in a configuration file or SQL query will render an application inoperable. Also, at the start, you may feel overwhelmed by the myriad complexities that confront you. There's the database, SQL syntax, relational database theory, new terms and technology, and the complexity of a multitiered application. To get a complex application to run correctly requires each piece of the application to work correctly. But fear not. At every step of the way I will show you how to compile (if necessary), configure, and run each piece of the puzzle.

When you finish this chapter you will be invincible! But don't stop here. Dive deeper into the topic by following your interests. There's much more to relational database design, ADO.NET, and the Microsoft Enterprise Library than what's covered here. Alas, there is always more to learn!

What You Are Going To Build

You, my friend, are going to build a multitiered, networked, data-driven, client-server application. The application will be used to track employee training. Users can create, edit, and delete employees as well as create, edit, and delete employee training records. Employee records stored in the database will include an employee picture, so you'll need to know how to store and retrieve image data.

The overall architectural diagram for the employee training server application is given in Figure 20-1.

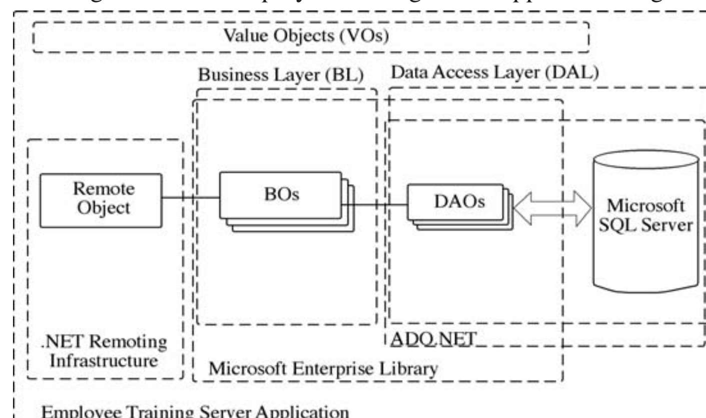


Figure 20-1: Employee Training Server Application Architecture

 © 2008 Rick Miller — All Rights Reserved

Referring to Figure 20-1 — the employee training server application comprises several application layers. These include the Business Layer (BL) where business objects (BOs) reside, and the Data Access Layer (DAL) where data access objects (DAOs) reside. The use of value objects (VOs) spans all application layers.

Different supporting Microsoft technologies come into play throughout the application. The Microsoft Enterprise Library can be used to support both business and data access layers, although in this chapter I am only using the Enterprise Library Data Access Application Block, which directly supports the data access layer. The .NET remoting infrastructure supports the remote object.

A *business object* is simply a class that contains the logic required to enforce the business rules of a particular application. However, try as one may to isolate *business rules* to business objects, they tend to creep into other parts of an application. For example, database design plays a key factor in what business rules can be enforced. (For example, what information about an employee is required and what information is optional?, etc.)

A business object will use the services of one or more data access objects. A *data access object* is a class whose job it is to interact with the database. As a rule, there is a one-to-one correspondence between data access objects and database tables. For example, an EmployeeDAO class would be responsible for interacting with the tbl_employee table in the database.

The data access layer uses the services of various classes, structures, interfaces, and enumerations provided by ADO.NET and the Microsoft Enterprise Library Data Access Application Block (DAAB). The DAAB, among other things, provides a DatabaseFactory class that is used to get a connection to the database. The DAAB also takes care of connection pooling to increase application performance when servicing multiple client connections.

The remote object supplies an interface used by remote client applications to interact with the server. The remote object uses the services provided by one or more business objects. As you know already from reading the previous chapter, a remote object requires the support of the .NET remoting infrastructure.

Referring again to Figure 20-1 — application layer dependencies flow from right to left. The business layer depends on the data access layer, and the remote object depends on the business layer. All layers depend on the value object layer, which spans all application layers.

Preliminaries

Before you move forward in this chapter, you must take the time to install Microsoft SQL Server Express Edition and the Microsoft Enterprise Library. You will also find it helpful to install the Microsoft SQL Server Management Studio Express Edition as well, but this is not strictly required to get the application up and running. The Management Studio application provides a robust GUI interface to your SQL Server database.

Installing SQL Server Express Edition

I use SQL Server 2005 Express Edition for the database in this chapter. (**Note:** You should be able to use SQL Server 2008 Express Edition with little or no problem.) Go to Microsoft's website, download the installation package, double click the installer executable file and follow the on-screen instructions. Installation starts with an overall system configuration check and the installation of some key components necessary for a smooth installation. If the configuration check goes well, you'll see a report similar to the one shown in Figure 20-2.

If you pass the system configuration check, you'll come to the feature selection dialog window, as is shown in Figure 20-3. At this point you can simply click the "Next >" button to proceed with the installation.

When you've finished installing SQL Server Express Edition, you can test the installation by opening a console window and entering the following command:

```
sqlcmd -S .\sqlexpress
```

This opens a connection to the default database. If all goes well you will get a line number. At the first line number "1>" enter the following SQL command:

```
select table_name from information_schema.tables
```

Press Enter. This will bring you to a second line number "2>" where you need to enter the following command:

```
go
```

Press Enter. The results you get should look similar to the output shown in Figure 20-4. To exit the SQL command prompt type the command "exit" at the line number, then press Enter.

© 2008 Rick Miller — All Rights Reserved

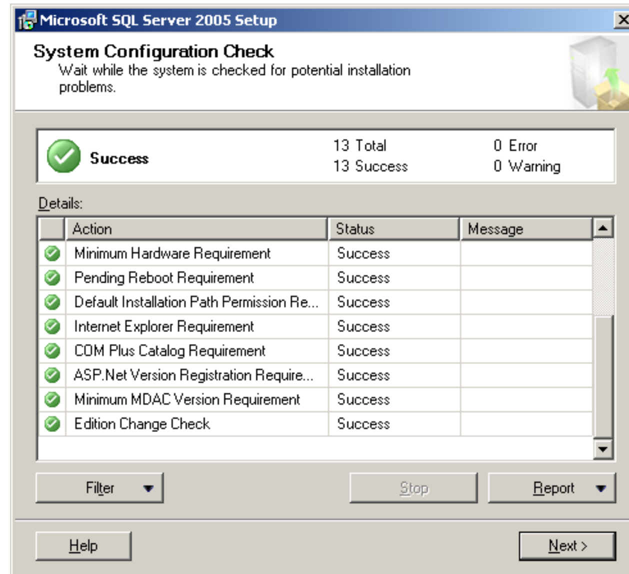


Figure 20-2: SQL Server System Configuration Check

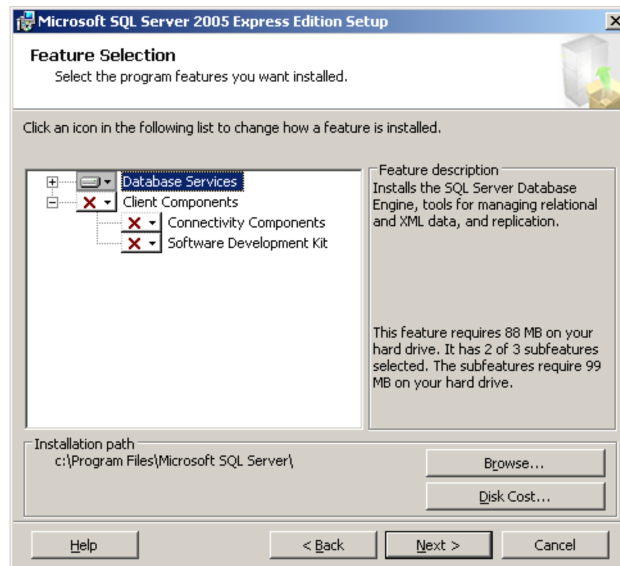


Figure 20-3: SQL Express Feature Selection Dialog

Installing Microsoft SQL Server Management Studio Express

You could do all your interaction with SQL Server Express via the SQL command utility, however, this can be cumbersome for beginners (and experienced developers too!). SQL Server Management Studio is a GUI-based application that makes it easy to manage and manipulate SQL Server databases.

You can download SQL Server Management Studio Express Edition from the same place you downloaded SQL Server Express. Follow the installation instructions and go with the default values. Installation is quick and painless. When you've finished, start SQL Server Management Studio by selecting All Programs->Microsoft SQL Server 2005->SQL Server Management Studio Express from the Start menu. This will display a login dialog window similar to the one shown in Figure 20-5.

Referring to Figure 20-5 — click the Connect button to connect to the designated Server name. If you have just installed SQL Server Express there will be only one server on the list! When you click the Connect button, you'll be logged into the server and your next window will look similar to the one shown in Figure 20-6.

 © 2008 Rick Miller — All Rights Reserved

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects>sqlcmd -S .\sqlexpress
1> select table_name from information_schema.tables
2> go
table_name

--------------------------------------------------------------------------------
--------
spt_fallback_db

spt_fallback_dev

spt_fallback_usg

spt_monitor

spt_values

MSreplication_options

(6 rows affected)
1> exit

C:\Documents and Settings\Rick\Desktop\Projects>_
```

Figure 20-4: Results of Testing SQL Server Express Edition Installation

Figure 20-5: Management Studio Login Dialog

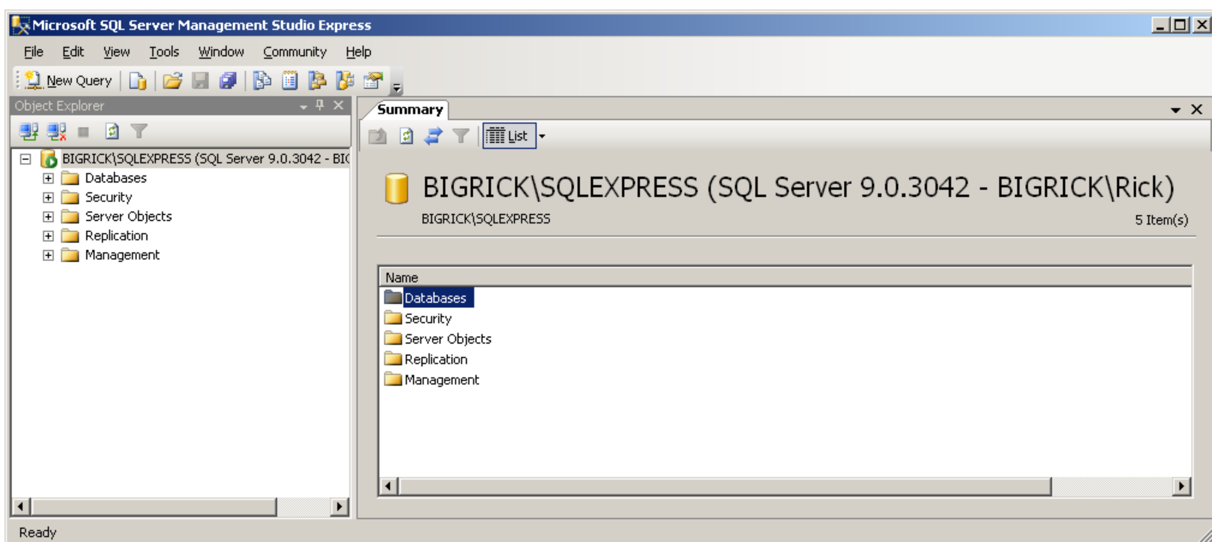


Figure 20-6: SQL Management Studio Main Window

© 2008 Rick Miller — All Rights Reserved

Installing Microsoft Enterprise Library

The final thing you need to install is the Microsoft Enterprise Library. In the interest of full disclosure, you don't need the enterprise library data application blocks to do ADO.NET programming. They just make ADO.NET programming easier to do. For the purposes of this chapter, the Enterprise Library Data Access Application Block is required.

Download the Microsoft Enterprise Library installer from the Microsoft Patterns and Practices site. Run the enterprise library installer. The second window you'll see will be the Custom Setup dialog window similar to the one shown in Figure 20-7.

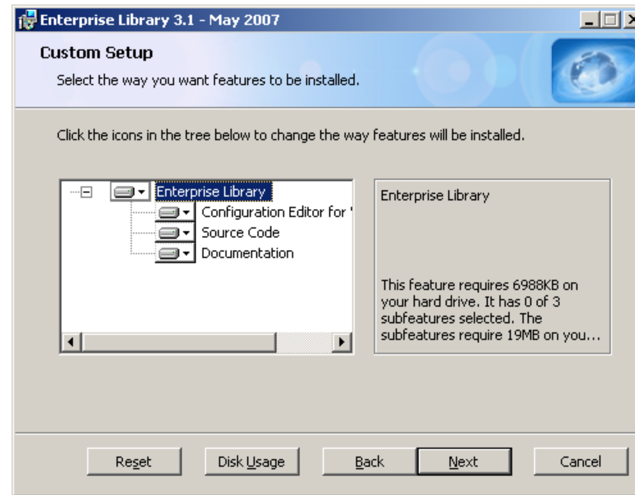
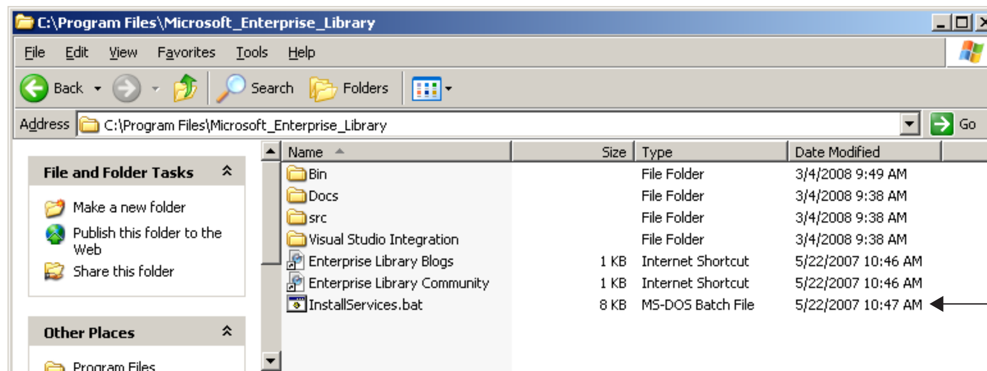


Figure 20-7: Enterprise Library Custom Setup Dialog

Referring to Figure 20-7 — accept the default installation by clicking the Next button. Installation will proceed fairly quick, however, you are not done just yet. When the installer completes, you'll need to build the libraries by navigating to the installation directory and double-clicking the InstallServices.bat file to compile and deploy the libraries (*i.e.*, the .dll files) Figure 20-8 shows my enterprise library installation directory (**Note:** I have changed the installation directory from its default name to Microsoft_Enterprise_Library.)



Double-click the InstallServices.bat file to compile and deploy the enterprise library .dll files.

Figure 20-8: Double-Click the InstallServices.bat File

Double-clicking the InstallServices.bat file will open a console window where you will see the libraries compiled automatically. When done, you should have a Bin directory with the enterprise library .dll files. You will be most concerned with the following three enterprise library .dll files:

- Microsoft.Practices.EnterpriseLibrary.Common.dll
- Microsoft.Practices.EnterpriseLibrary.Data.dll
- Microsoft.Practices.EnterpriseLibrary.ObjectBuilder.dll

 © 2008 Rick Miller — All Rights Reserved

A Simple Test Application

This section presents a short, simple test application that will make sure you've got everything installed correctly. Don't proceed past this point until you get this application to run. When you're successful, you can rest assured you've got this chapter half licked!

Example 20.1 gives the code for a short application named SimpleConnection that uses a DatabaseFactory to create a Database object, and then executes a simple SQL command against that database.

20.1 SimpleConnection.cs

```
1    using System;
2    using System.Data;
3    using System.Data.Common;
4    using System.Data.Sql;
5    using System.Data.SqlClient;
6
7    using Microsoft.Practices.EnterpriseLibrary.Common;
8    using Microsoft.Practices.EnterpriseLibrary.Data;
9    using Microsoft.Practices.EnterpriseLibrary.Data.Sql;
10
11   public class SimpleConnection {
12     public static void Main(){
13       Console.WriteLine("Simple Connection!");
14       Database database = DatabaseFactory.CreateDatabase();
15       Console.WriteLine("Database created!");
16       DbCommand command = database.GetSqlStringCommand("select table_name from information_schema.tables");
17       IDataReader reader = database.ExecuteReader(command);
18       while(reader.Read()){
19         Console.WriteLine(reader.GetString(0));
20       }
21     } // end Main()
22   } // end class definition
```

Referring to Example 20.1 — note first the namespaces required. On line 14, the DatabaseFactory.CreateDatabase() method is called to create a Database object. At this point you should be wondering from where on earth does the DatabaseFactory class get the information required to create the Database object? The answer is — from a configuration file, which you'll see shortly.

On line 16, the Database object's GetSqlStringCommand() method is used to create a DbCommand object. The string used as an argument to the GetSqlStringCommand() method is a short SQL select statement, just like the one you used earlier to test the installation of SQL Server Express. The command is executed via a call to the Database object's ExecuteReader() method using the reference to the newly created Command object as an argument. It returns an IDataReader object which you use to access the query results in the body of the `while` loop. The output of this program will be a list of table names like that obtained originally in Figure 20-4.

Example 20.2 shows the contents of the simpleconnection.exe.config file.

20.2 simpleconnection.exe.config

```
1    <configuration>
2      <configSections>
3        <section name="dataConfiguration"
4                 type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings,
5                 Microsoft.Practices.EnterpriseLibrary.Data,
6                 Version=3.1.0.0, Culture=neutral,
7                 PublicKeyToken=b03f5f7f11d50a3a" />
8      </configSections>
9      <dataConfiguration defaultDatabase="Connection String" />
10     <connectionStrings>
11       <add name="Connection String"
12                  connectionString="Data Source=(local)\SQLEXPRESS;Initial Catalog=master;
13         Integrated Security=True"
14          providerName="System.Data.SqlClient" />
15     </connectionStrings>
16   </configuration>
```

Referring to Example 20.2 — the configuration file provides database connection string information. You create these configuration files with the help of the Enterprise Library Configuration tool, which you'll find in the enterprise library's installation directory. A screen shot showing the tool in action is shown in Figure 20-9. At this point it would be easier for you to either download this configuration file from the pulpfreepress.com website or create it manually by copying it from the example above.

Alright — you have the SimpleConnection.cs file and the simpleconnection.exe.config file. Before you compile the application, you'll need to copy the three required Enterprise Library dll files into your project directory. Your project directory should look similar to the one shown in Figure 20.10.

© 2008 Rick Miller — All Rights Reserved

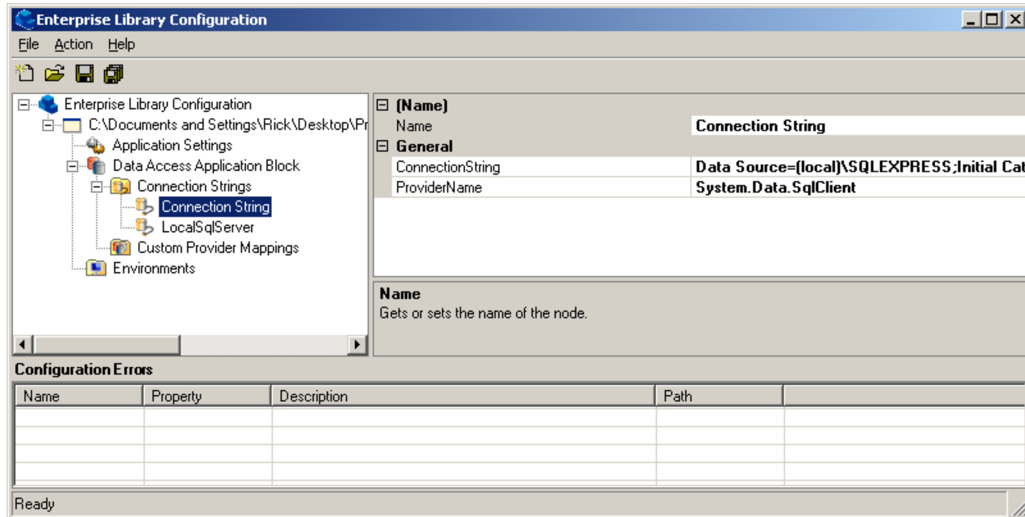


Figure 20-9: Enterprise Library Configuration File Creation Tool

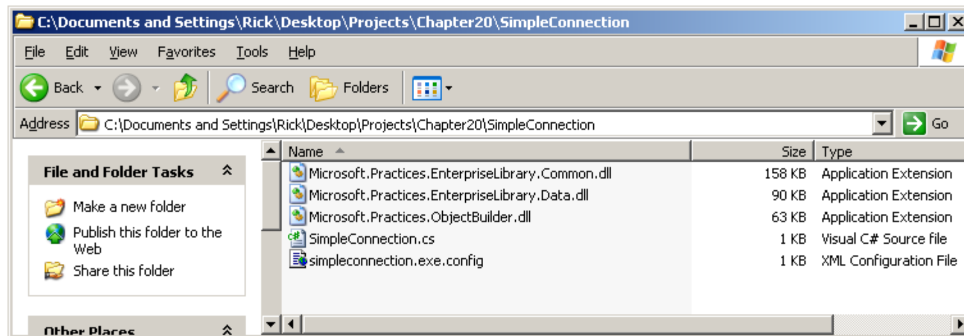


Figure 20-10: Contents of the SimpleConnection Project Directory Before Compiling

To compile this application, open a command console window, change to the project directory, and enter the following compiler command:

```
csc /r:Microsoft.Practices.EnterpriseLibrary.Data.dll;Microsoft.Practices.EnterpriseLibrary.Common.dll /
lib:"C:\Program Files\Microsoft_Enterprise_Library\Bin" *.cs
```

Note that this is all on one line and that there is no hyphen in the word Enterprise. Also note that I have changed the name of my enterprise library installation directory to Microsoft_Enterprise_Library, so on my computer the enterprise library .dll files are located in the Microsoft_Enterprise_Library\Bin directory as I've given in the `/lib` compiler switch above.

When you've entered this command, press Enter and cross your fingers. If all goes well it will compile. If not, you'll need to retrace your steps to ensure you've installed the database and the enterprise library files correctly.

Finally, run the application by typing simpleconnection at the command prompt. You should see an output similar to what is shown in Figure 20-11.

Introduction To Relational Databases And SQL

In this section I will show you how to create and manipulate data contained in a relational database using Structured Query Language (SQL). You'll also learn how to create SQL scripts to automate the execution of complex queries and other commands.

 © 2008 Rick Miller — All Rights Reserved

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\SimpleConnection>simpleconnection
Simple Connection!
Database created!
spt_fallback_db
spt_fallback_dev
spt_fallback_usg
spt_monitor
spt_values
MSreplication_options

C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\SimpleConnection>_
```

Figure 20-11: Results of Running the SimpleConnection Application

Terminology

A *database management system* (DBMS) is a software application that stores data in some form or another and provides a suite of software components that allows users to create, manipulate, and delete the data. The term *database* refers to a related collection of data. A DBMS may contain one or more databases. The term database is often used interchangeably to refer both to a DBMS and to the databases it contains. For example, a colleague is more likely to ask you "What type of database are you going to use?" rather than "What type of database management system are you going to use?"

A *relational database* stores data in relations referred to as *tables*. A *relational database management system* (RDBMS) is a software application that allows users to create and manipulate relational databases. Popular RDBMS systems that I'm personally familiar with include Oracle, MySQL, and Microsoft SQL Server, but there exist many more.

A table is composed of *rows* and *columns*. Each column has a *name* and an associated database *type*. For example, a table named tbl_employee may have a column named FirstName with a type of varchar(50). (*i.e.,* A variable-length character field with a 50 character limit.) Each row is an instance of data stored in the table. For example, the tbl_employee table might contain any number of employee entries, with each entry occupying a single row in the table.

In most cases it is desirable to be able to uniquely identify each row of data contained within a table. To do this, one or more of the table columns must be designated as the *primary key* for that table. The important characteristic of a primary key is that its value must be unique for each row.

The power of relational databases derives from their ability to dynamically create associations between different tables. One table can be related to another table by the implementation of a *foreign key*. The primary key of one table serves as the foreign key in the related table, as Figure 20-12 illustrates.

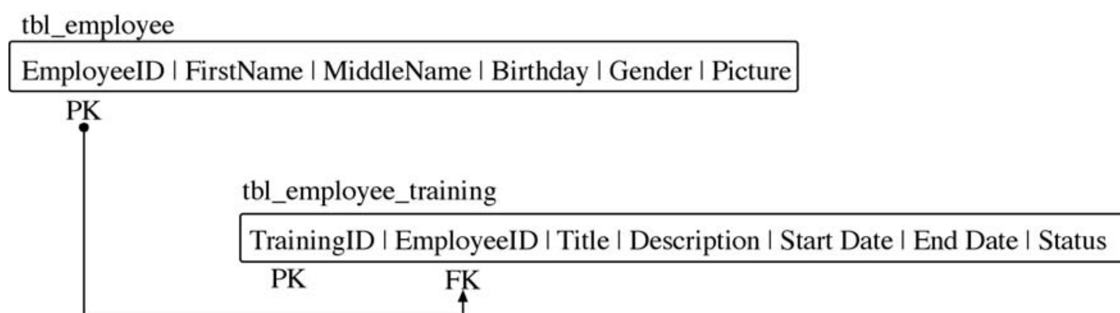


Figure 20-12: The Primary Key of One Table Can Serve as the Foreign Key in a Related Table

Referring to Figure 20-12 — the EmployeeID column serves as the primary key for tbl_employee. An EmployeeID column in tbl_employee_training serves as a foreign key for that table. In this manner, a relationship has been established between tbl_employee and tbl_employee_training. These tables can now be manipulated together to extract meaningful data regarding employees and the training they have taken. A table can be related to multiple tables by the inclusion of multiple foreign keys.

Primary keys and foreign keys can be used together to enforce *referential integrity*. For example, you should not be able to insert a new row into tbl_employee_training unless the EmployeeID foreign key value you are trying to

© 2008 Rick Miller — All Rights Reserved

insert already exists as a primary key in tbl_employee. Also, what should happen when an employee row is deleted from tbl_employee? A *cascade delete* can automatically delete any related records in tbl_employee_training. When an employee row is deleted from tbl_employee, any rows in tbl_employee_training with a matching foreign key will also be deleted.

Structured Query Language (SQL)

SQL is used to create, manipulate, and delete relational database objects and the data they contain. Although SQL is a standardized database language, each RDBMS vendor is free to add extensions to the language, which essentially renders the language non-portable between different database products. What this means to you is that while the examples I present in this section will work with Microsoft SQL Server, they may not work with Oracle, MySQL, or whatever relational database system you're familiar with. This holds true especially for SQL's Data Definition Language commands, which we will cover shortly.

SQL comprises three sub-languages, which group commands according to functionality: Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL). In this section I will focus on the use of DDL and DML.

Data Definition Language (DDL)

The DDL includes the `create`, `use`, `alter`, and `drop` commands. Let's use a few of these commands to set up the employee training database that will be used to store data for the employee training application. Before we begin, open SQL Server Management Studio and take a look at the default databases SQL Server provides upon installation.

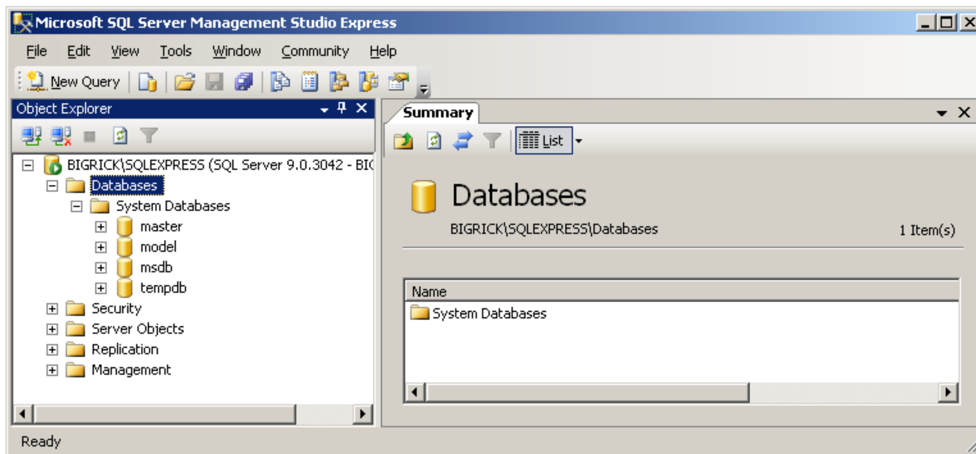


Figure 20-13: SQL Server's Default Databases

Referring to Figure 20-13 — there are four databases installed by default. These include *master*, *model*, *msdb*, and *tempdb*. Of these four, the *master* database is most important. It contains data necessary to startup and run SQL Server. Ordinarily, you will not directly interface with or manipulate the master database, but you will need to use it every once in a while with the `use` command as you will see shortly.

Creating The EmployeeTraining Database

Let's now create the EmployeeTraining database with the help of the `create` command. You could create the database using Management Studio, but I want to show you how to do it using the SQL command utility and then with the help of an SQL script file.

First, open a command window and start the SQL command utility with the following command:

```
sqlcmd -S .\sqlexpress
```

On the first numbered line enter the following command:

```
create database EmployeeTraining
```

 © 2008 Rick Miller — All Rights Reserved

Press Enter, and on the second numbered line enter the following command:

```
go
```

Press Enter. Your command window should look similar to Figure 20-14. Check that the database exists by open-

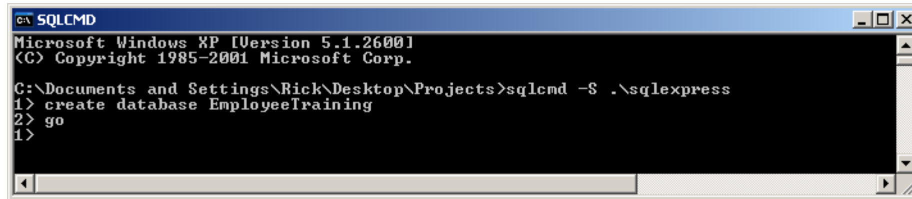


Figure 20-14: Creating EmployeeTraining Database with SQL Command Utility

ing Management Studio and taking a look. You should see something similar to Figure 20-15.

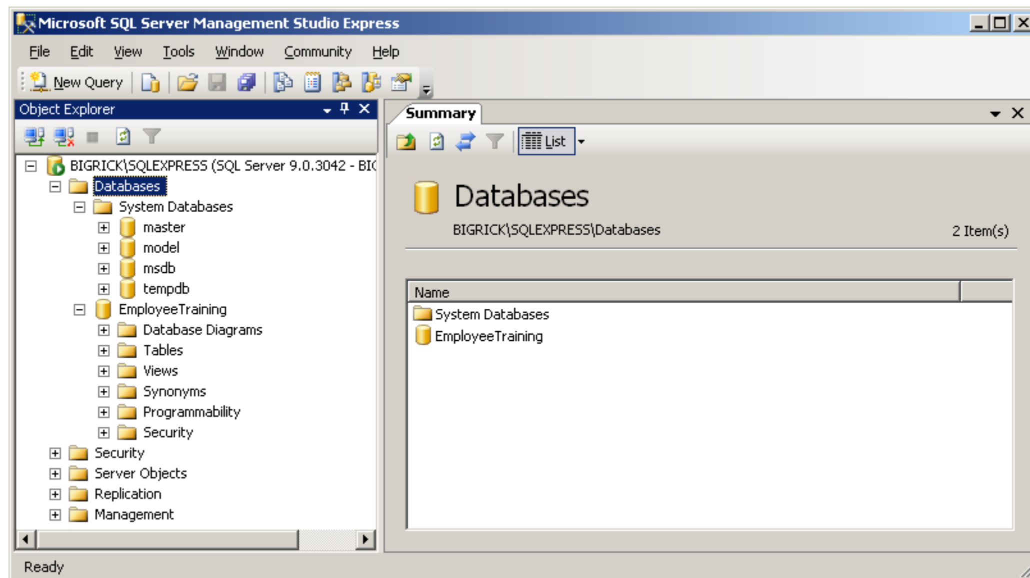


Figure 20-15: Checking on the Existence of the EmployeeTraining Database

Creating A Database With A Script

Alright, now that you've done this via the SQL command utility line-by-line, I want to show you how to drop the database and create it with a script. Open your favorite text editor and create a file named "create_database.sql" and enter the code shown in Example 20.3

20.3 create_database.sql

```
1     use master
2     drop database EmployeeTraining
3     go
4
5     create database EmployeeTraining
6     go
```

Referring to Example 20.3 — one line 1, the `use` command switches to the master database context. The `drop` command on line 2 drops the EmployeeTraining database. The `go` command on line 3 executes the previous two lines. On line 5, the `create` command creates the EmployeeTraining database. The `go` command is used again on line 6 to execute line 5.

Save the create_database.sql file in a folder named "scripts". In fact, now would be a good time to create a project folder for the employee training application. I recommend two folders: one named "client", the other named "server". Create the scripts folder in the server folder.

To execute the create_database.sql script, change to the scripts folder and enter the following command:

```
sqlcmd -S .\sqlexpress -i create_database.sql
```


© 2008 Rick Miller — All Rights Reserved

Press Enter. If all goes well, you'll see an output similar to that shown in Figure 20-16. As you can see from look-

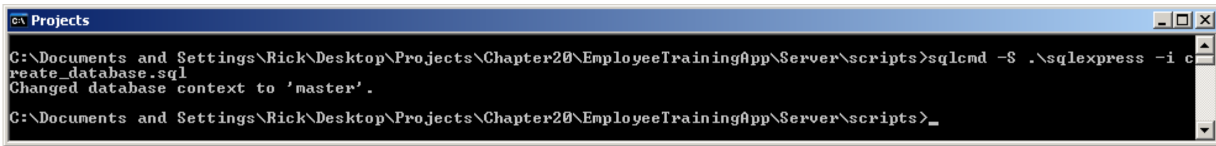


Figure 20-16: Results of Executing the create_database.sql Script

ing at Figure 20-16, there's not much output, only one line indicating the database context changed to master. Open Management Studio and verify once again that the EmployeeTraining database exists. It's now time to create the tables we'll use to store the employee training application data.

Creating Tables

The `create` command is used to create the tables we'll need to store data for employees and their training. Now, while you could create the tables via the SQL command utility line-by-line, that method is error-prone and hard to edit. It's much easier to create a script to do the work for you. Example 20.4 gives the first version of a database script named "create_tables.sql" that contains the SQL code required to create a table named "tbl_employee".

20.4 create_tables.sql (1st version)

```
1    use EmployeeTraining
2
3    drop table tbl_employee
4    go
5
6    create table tbl_employee (
7       EmployeeID uniqueidentifier not null primary key,
8       FirstName varchar(50) not null,
9       MiddleName varchar(50) not null,
10      LastName varchar(50) not null,
11      Birthday datetime not null,
12      Gender varchar(1) not null,
13      Picture varbinary(MAX) null
14   )
15   go
```

Referring to Example 20.4 — it's imperative that this script executes in the EmployeeTraining database, and that's the purpose of the `use` command on line 1. Line 3 drops the tbl_employee table, if it exists. It certainly will *not* exist the first time you execute the script, so you'll see an error message stating that fact. You can safely ignore that message. The create command starting on line 6 creates the tbl_employee table. The tbl_employee table contains seven columns named *EmployeeID*, *FirstName*, *MiddleName*, *LastName*, *Birthday*, *Gender*, and *Picture*. Each column has a corresponding database type. Most are of the variable length character type varchar(*n*) where *n* specifies the maximum number or characters the column can contain. The EmployeeID column is of type *uniqueidentifier* which has been designated as the table's primary key column. The Birthday column is of type datetime, and the Picture column is a variable length binary column set to varbinary(MAX). All columns except Picture must contain data when a row is created. This is specified with the `not null` constraint. (A *constraint* is a rule placed on a column or table meant to enforce data integrity.)

In this example application the tbl_employee table is fairly simple and straightforward. I may, in the not to distant future, regret the decision to put the employee picture in the tbl_employee table, but for now that's where I'm putting it!

To run this script save it in the scripts folder, open a command window, change to the scripts folder, and enter the following command-line command:

```
sqlcmd -S .\sqlexpress -i create_tables.sql
```

The results obtained from executing this script on my machine are shown in Figure 20-17.

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\scripts>sqlcmd -S .\
sqlexpress -i create_tables.sql
Changed database context to 'EmployeeTraining'.
Msg 3701, Level 11, State 5, Server BIGRICK\SQLEXPRESS, Line 3
Cannot drop the table 'tbl_employee', because it does not exist or you do not have permission.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\scripts>_
```

Figure 20-17: Results of Executing create_tables.sql Database Script

SQL Server Database Types

Table 20-1 lists the MS SQL Server database types and their associated value ranges and usage.

Type Category	Data Type	Value Range	Usage
Exact Numeric	bigint	-2^{63} to $2^{63}-1$	Use to store large integral values.
Exact Numeric	int	-2^{31} to $2^{31}-1$	Use to store medium-sized integral values.
Exact Numeric	smallint	-2^{15} to $2^{15}-1$	Use to store small integral values.
Exact Numeric	tinyint	0 to 255	Use to store really small integral values.
Exact Numeric	bit	0, 1, or null	Stores 1 or 0
Exact Numeric	decimal	$-10^{38}+1$ to $10^{38}-1$	decimal(p, s) where p is precision and s is scale. P is the maximum total number of decimal digits that can be stored both to the left and right of the decimal point. The range of p is 1 - 38 with 18 as the default. Scale is the maximum number of decimal digits that can be stored to the right of the decimal point. The range of s varies from 0 - p. (Example decimal(24, 6) would specify 24 total digits with 6 to the right of the decimal point.)
Exact Numeric	numeric	$-10^{38}+1$ to $10^{38}-1$	numeric is equivalent to decimal
Exact Numeric	money	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	Large monetary or currency values. Use to hold the value of US national debt.
Exact Numeric	smallmoney	-214,748.3648 to 214,748.3647	Small monetary or currency values.
Approximate Numerics	float	-1.79^{308} to -2.23^{-308}, 0, and 2.23^{-308} to 1.79^{308}	float(n) where n is the number of bits used to store the mantissa. n must be a value between 1 - 53. Default value of n is 53.
Approximate Numerics	real	-3.40^{38} to -1.18^{-38}, 0 and 1.18^{-38} to 3.4^{38}	Equivalent to float
Date and Time	datetime	1 January 1753 through 31 December 9999	Holds a large date and time range.

Table 20-1: SQL Server Data Types

© 2008 Rick Miller — All Rights Reserved

Type Category	Data Type	Value Range	Usage
Date and Time	smalldatetime	1 January 1900 through 6 June 2079	Holds a smaller date and time range.
Character Strings	char	1 - 8000 fixed length bytes	Holds fixed length character values.
Character Strings	varchar	1 - 8000 variable length bytes or varchar(MAX) holds 2^{31}-1 bytes	Holds variable length character strings varchar(n) where n specifies max length.
Character Strings	text	DO NOT USE	Will be removed from future versions of SQL Server
Unicode Character Strings	nchar	1 - 4000 fixed length unicode characters	Holds fixed length unicode character strings.
Unicode Character Strings	nvarchar	1 - 4000 variable length unicode characters or nvarchar(MAX) holds 2^{31}-1 bytes	Holds variable length unicode character strings. nvarchar(n) where n specifies max length.
Unicode Character Strings	ntext	DO NOT USE	Will be removed from future versions of SQL Server
Binary Strings	binary	1 - 8000 fixed length binary data	Holds fixed length binary data.
Binary Strings	varbinary	1 - 8000 variable length binary data or varbinary(MAX) holds 2^{31}-1 bytes	Holds variable length binary data. varbinary(n) where n specifies max length.
Binary Strings	image	DO NOT USE	Will be removed from future versions of SQL Server
Other	cursor	cursor reference	Holds variables or stored procedure output parameters that contain a reference to a cursor.
Other	sql_variant	int, binary, and char	Stores values of various data types.
Other	table	result set	Stores a result set for later processing.
Other	timestamp	Automatically generated unique binary number	Used to version-stamp table rows. Does not preserve a date or a time.
Other	uniqueidentifier	A 16-byte Globally Unique Identifier (GUID)	Used to hold GUID strings.
Other	xml	2 gigabytes	Holds XML data.

Table 20-1: SQL Server Data Types

Referring to Table 20-1 — note that three database types have been deprecated and will, at some point in the future, be dropped from SQL Server. These have been highlighted with light grey shading. Now, while this may or may not happen for a long, long time, it's still a good idea to shy away from using the deprecated types when writing new code.

Data Manipulation Language (DML)

Now that you've created the EmployeeTraining database and added to it the tbl_employee table, it's time to learn how to use SQL's Data Manipulation Language to add, manipulate, and delete tbl_employee data. There are four DML commands: `insert`, `select`, `update`, and `delete`. First things first! Let's create a script to insert some test data into the tbl_employee table. I'll then show you how to manipulate that data with the other three commands.

© 2008 Rick Miller — All Rights Reserved

Using The Insert Command

Example 20.5 gives the code for a database script named "create_test_data.sql" that inserts one row of test data into the tbl_employee table.

20.5 create_test_data.sql

```
1    use EmployeeTraining
2    go
3
4    insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
5    values (newid(), 'Rick', 'Warren', 'Miller', '3/13/1961', 'M')
6    go
```

Referring to Example 20.5 — line 1 is necessary to ensure we are using the correct database, which in this case is EmployeeTraining. Line 4 contains the first part of the insert statement. With the `insert` statement you specify into which table you want to insert data, and list each column that will receive data in the parentheses. The order of the columns you specify is important because the order of the actual values you insert, shown here on line 5, must match the order in which you listed your columns. In this example, I am inserting data into the employeeid, firstname, middlename, lastname, birthday, and gender columns only. Remember, these columns MUST contain data because of their NOT NULL constraint. It's ok not to insert data into the picture column because that column is allowed to contain a null value. (**Note:** If you want to insert more than one row of test data simply add another insert statement to the script below line 5.)

To execute this script, open a command window, change to the scripts folder, and enter the following command:

```
sqlcmd -S .\sqlexpress -i create_test_data.sql
```

Then press Enter. You should see a result similar to that shown in Figure 20-18.

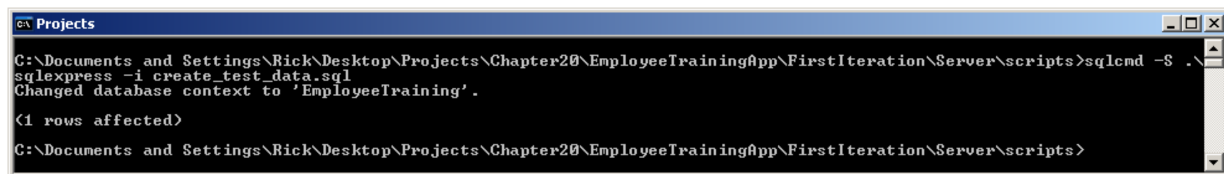


Figure 20-18: Results of Running create_test_data.sql Database Script

Using The Select Command

The `select` command is used to write database queries (*i.e.,* select statements) that return data. A `select` statement contains several clauses, most of which are optional. The following code fragment shows a simple `select` statement that gets all the data contained in all the columns of the tbl_employee table:

```
select * from tbl_employee
```

To execute this `select` statement, open the SQL command utility by typing the following command-line command:

```
sqlcmd -S .\sqlexpress
```

At the first numbered line enter the following command:

```
use employeetraining
```

Press Enter, then enter `go` and press Enter again. On the first numbered line enter the `select` command given above and press Enter. On the next numbered line enter the `go` command and press Enter. Your results should look similar to Figure 20-19.

Referring to Figure 20-19 — the output is a little bunched up but you can pick out the column headings and their associated data.

You can limit the number of columns a select statement returns by specifying exactly which columns you want when you enter the select statement, as the following code fragment shows:

```
select firstname, middlename, lastname from tbl_employee
```

Try executing this statement in the SQL command utility. Your results should look similar to those shown in Figure 20-20.

Up to this point I've only been using one required `select` statement `from` clause to specify the table from which to get the data. The following `select` statement adds an optional `where` clause to limit the data returned:

```
select firstname, lastname from tbl_employee where lastname='Bishop'
```


© 2008 Rick Miller — All Rights Reserved

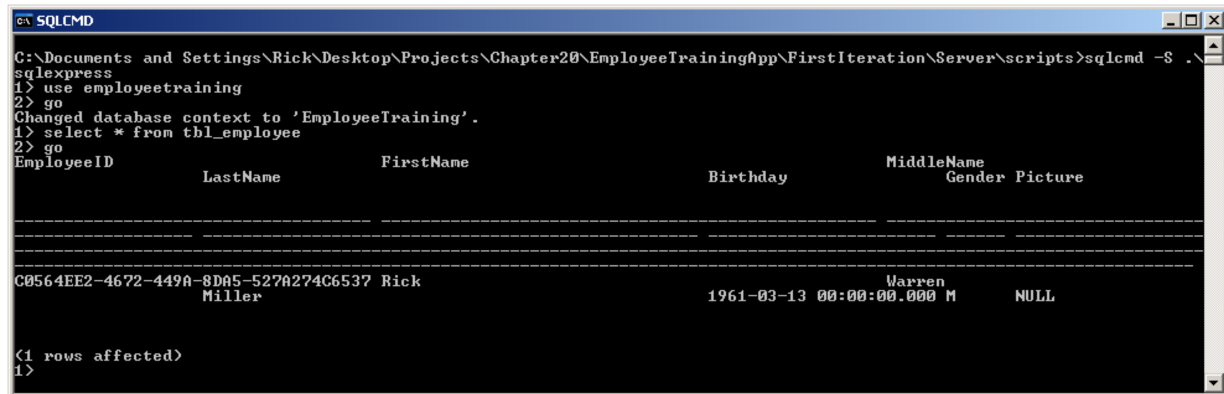


Figure 20-19: Results of Executing a Simple Select Statement

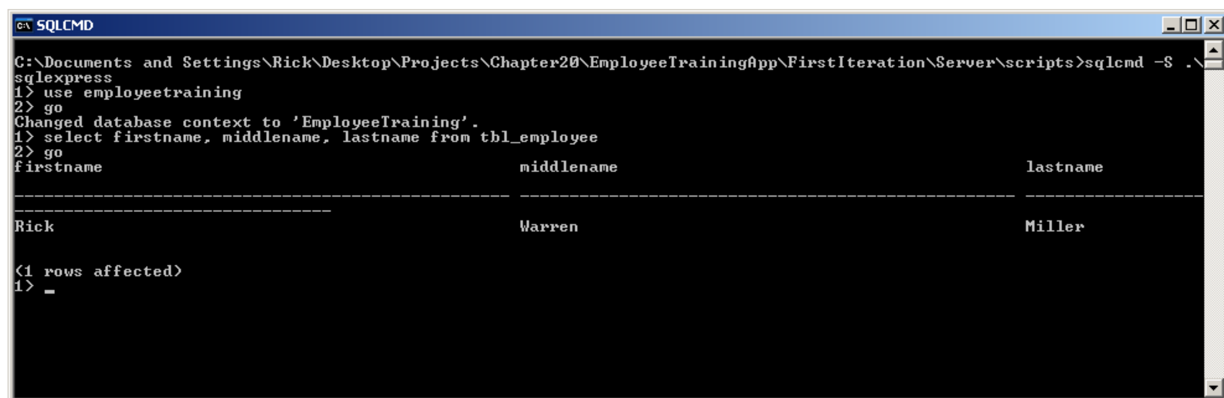


Figure 20-20: Selecting Specific Rows with select Statement

As you may have guessed, if you entered this query in the employeetraining database, you'd get no results because nobody by the last name of Bishop has been entered into the tbl_employee table. Let's modify the create_test_data.sql script to add some more test data. Example 20.6 gives the modified script.

20.6 create_test_data.sql (Mod 1)

```
1    use EmployeeTraining
2    go
3
4    insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
5    values (newid(), 'Rick', 'Warren', 'Miller', '3/13/1961', 'M')
6    go
7    insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
8    values (newid(), 'Steve', 'Jacob', 'Bishop', '2/10/1942', 'M')
9    go
10   insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
11   values (newid(), 'Coralie', 'Sarah', 'Powell', '10/10/1974', 'F')
12   go
13   insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
14   values (newid(), 'Kyle', 'Victor', 'Miller', '8/25/1986', 'M')
15   go
16   insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
17   values (newid(), 'Patrick', 'Tony', 'Condemi', '4/17/1961', 'M')
18   go
19   insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
20   values (newid(), 'Dana', 'Lee', 'Condemi', '11/1/1965', 'F')
21   go
```

Referring to Example 20.6 — notice how the `go` command must be issued after each `insert` statement. Run this script to insert the extra data. (**Note:** You may want to run the create_tables.sql script to start clean! Don't you just love database scripts!) Your results should look similar to Figure 20-21.

Now, start the SQL command utility, change to the employeetraining database, and enter the following `select` statement:

```
select firstname, lastname
from tbl_employee
```

 © 2008 Rick Miller — All Rights Reserved

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\scripts>sqlcmd -S .\
sqlexpress -i create_test_data.sql
Changed database context to 'EmployeeTraining'.

(1 rows affected)

(1 rows affected)

(1 rows affected)

(1 rows affected)

(1 rows affected)

(1 rows affected)

C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\scripts>_
```

Figure 20-21: Inserting More Test Data with the create_test_data.sql Database Script

```
where lastname='Miller'
go
```

This is how you will normally see a `select` statement used, with each clause appearing on separate lines. Note that the `go` command is not part of SQL, but rather how the SQL statement is executed in SQL Server. The results you get from executing this query should look similar to those shown in Figure 20-22.

```
SQLCMD

C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\scripts>sqlcmd -S .\
sqlexpress
1> use employeetraining
2> go
Changed database context to 'EmployeeTraining'.
1> select firstname, lastname
2> from tbl_employee
3> where lastname='Miller'
4> go
firstname                                          lastname
-------------------------------------------------- --------------------------------------------------
Rick                                               Miller
Kyle                                               Miller

(2 rows affected)
1> _
```

Figure 20-22: Results of Limiting Data Returned from select Statement with *where* Clause

Referring to Figure 20-22 — the query returned two employees with the last name Miller. If you did not run the create_tables.sql script before running the modified create_test_data.sql script, you would see three employees in the results because there would be two Rick Millers in the database.

Try this query:

```
select firstname, lastname
from tbl_employee
where gender='F' or firstname='Kyle'
go
```

This should return three rows as is shown in Figure 20-23.

```
SQLCMD
1> select firstname, lastname
2> from tbl_employee
3> where gender='F' or firstname='Kyle'
4> go
firstname                                          lastname
-------------------------------------------------- --------------------------------------------------
Kyle                                               Miller
Dana                                               Condemi
Coralie                                            Powell

(3 rows affected)
1>
```

Figure 20-23: Results of Executing the Previous Query

Using The Update Command

Data within a table can be changed with the SQL `update` command. For example, if you wanted to change the employee Coralie Powell's last name to Miller, you would use the following `update` statement:

```
update tbl_employee
```

 © 2008 Rick Miller — All Rights Reserved

```
set lastname = 'Miller'
where firstname = 'Coralie'
go
```

The `update` statement begins by specifying the name of the table to which the update applies. The `set` clause on the second line specifies one or more columns within that table and their new values. The `where` clause is used to specify to which row in the table the update applies. In this case the employee whose first name is "Coralie" will have her name changed from "Powell" to "Miller". (**Note:** If you had more than one employee with the first name "Coralie", this statement would change all their last names to Miller. To isolate the correct Coralie you'd have to use her EmployeeID in the `where` clause.) Figure 20-24 illustrates the use of the previous `update` statement.

```
SQLCMD
C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\scripts>sqlcmd -S .\
sqlexpress
1> use employeetraining
2> go
Changed database context to 'EmployeeTraining'.
1> select firstname, lastname
2> from tbl_employee
3> go
firstname                                                   lastname
----------------------------------------------------------- ------------------------------------------------------------
Dana                                                        Condemi
Rick                                                        Miller
Steve                                                       Bishop
Kyle                                                        Miller
Coralie                                                     Powell
Patrick                                                     Condemi

(6 rows affected)
1> update tbl_employee
2> set lastname = 'Miller'
3> where firstname = 'Coralie'
4> go

(1 rows affected)
1> select firstname, lastname
2> from tbl_employee
3> go
firstname                                                   lastname
----------------------------------------------------------- ------------------------------------------------------------
Dana                                                        Condemi
Rick                                                        Miller
Steve                                                       Bishop
Kyle                                                        Miller
Coralie                                                     Miller
Patrick                                                     Condemi

(6 rows affected)
1> _
```

Figure 20-24: Changing Coralie Powell's Last Name to Miller with the Update Statement

Using The Delete Command

The `delete` command is used to delete one or more rows from a table. The following `delete` statement removes from the tbl_employee table all employees whose last names equal "Miller":

```
delete from tbl_employee
where lastname = 'Miller'
go
```

The results of executing this statement are shown in Figure 20-25.

```
SQLCMD
C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\scripts>sqlcmd -S .\
sqlexpress
1> use employeetraining
2> go
Changed database context to 'EmployeeTraining'.
1> delete from tbl_employee
2> where lastname = 'Miller'
3> go

(3 rows affected)
1> select firstname, lastname
2> from tbl_employee
3> go
firstname                                                   lastname
----------------------------------------------------------- ------------------------------------------------------------
Dana                                                        Condemi
Steve                                                       Bishop
Patrick                                                     Condemi

(3 rows affected)
1> _
```

Figure 20-25: Deleting all Employees whose Last Names = "Miller"

© 2008 Rick Miller — All Rights Reserved

Quick Review

Relational databases hold data in *tables*. Table *columns* are specified to be of a particular *data type*. Table data is contained in *rows*. Structured Query Language (SQL) is used to *create*, *manipulate*, and *delete* relational database objects and data. SQL contains three sub-languages: Data Definition Language (DDL) which is used to create databases, tables, views, and other database objects; Data Manipulation Language (DML) which is used to create, manipulate, and delete the data contained within a database; and Data Control Language (DCL) which is used to grant or revoke user rights and privileges on database objects.

Different database makers are free to extend SQL to suit their needs so there's no guarantee of SQL portability between different databases.

One or more table columns can be designated as a *primary key* whose value is unique for each row inserted into that table. Related tables can be created by including the primary key of one table as a *foreign key* in the related table.

Complex SQL Queries

In this section I want to show you how to use SQL to manipulate data in multiple tables. Along the way you will learn how to use a *foreign key* to create a related table, and how to use the `from` clause to *join* related tables together in a `select` statement.

Creating A Related Table With A Foreign Key

It's time now to add another table to the employeetraining database. Where do you think would be a good place to put this table's `create` statement code? If you guessed the create_tables.sql script you're right!

20.7 create_tables.sql (Mod 1)

```
1    use EmployeeTraining
2
3    alter table tbl_employee_training drop constraint fk_employee
4    go
5
6    drop table tbl_employee
7    go
8
9    create table tbl_employee (
10      EmployeeID uniqueidentifier not null primary key,
11      FirstName varchar(50) not null,
12      MiddleName varchar(50) not null,
13      LastName varchar(50) not null,
14      Birthday datetime not null,
15      Gender varchar(1) not null,
16      Picture varbinary(max) null
17   )
18   go
19
20   drop table tbl_employee_training
21   go
22
23   create table tbl_employee_training (
24      TrainingID int not null identity(1,1) primary key,
25      EmployeeID uniqueidentifier not null,
26      Title varchar(200) not null,
27      Description varchar(500) not null,
28      StartDate datetime null,
29      EndDate datetime null,
30      Status varchar(25)
31   )
32   go
33
34   alter table tbl_employee_training
35   add constraint fk_employee
36   foreign key (EmployeeID)
37   references tbl_employee (EmployeeID) on delete cascade
38   go
```

Referring to Example 20.7 — the `create` statement for the tbl_employee_training table begins on line 23. It's preceded by the `drop` statement on line 20. The table's primary key is named TrainingID. The primary key value, in this case an integer, will be automatically generated when a record is inserted into the table and incremented by 1.

 © 2008 Rick Miller — All Rights Reserved

This behavior is obtained with the identity(1,1) entry specification. The first value is the identity seed, the second is the increment value.

Note that the tbl_employee_training table has a column named EmployeeID, which is the same type as the EmployeeID column in the tbl_employee table. However, this alone does not establish the foreign key relationship between that column and the one in the tbl_employee table. The *foreign key constraint* is created with the `alter` statement beginning on line 34. Note that the name of the foreign key constraint is fk_employee. (You could name it anything you like.) Having the foreign key constraint named in this manner allows you to drop the constraint before you drop the tbl_employee table. If you don't drop the fk_employee constraint before trying to drop the tbl_employee table you'll get an error. That's why it's necessary to put the `alter` statement on line 3.

To run this script, change to the scripts directory and enter the following command at the command-line:

```
sqlcmd -S .\sqlexpress -i create_tables.sql
```

The first time you run this script you'll get several errors saying the fk_employee constraint and tbl_employee_training table do not exist. When you run it a second time you will not receive those errors. After you run the script, verify the existence of the tbl_employee_training table by opening SQL Server Management Studio as is shown in Figure 20-26.

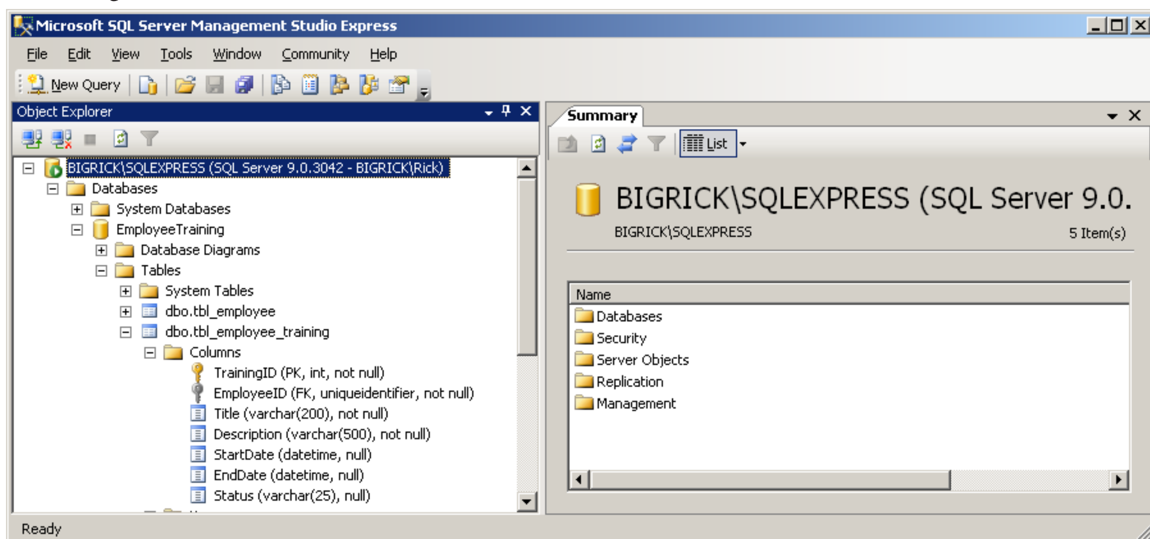


Figure 20-26: Verifying the Creation of the tbl_employee_training Table

Inserting Test Data Into The tbl_employee_training Table

You'll want to insert some test data into the tbl_employee_training table and to do this you'll need to make several modifications to the create_test_data.sql script. But first, run the script as-is to insert test data into the tbl_employee table. You need to do this so that you can get a valid GUID for each employee. To do this, run the script, then enter the following command in the SQL command utility: (Don't forget to change to the employeetraining database first!)

```
select employeeid from tbl_employee
go
```

Figure 20-27 shows this statement being executed in the SQL command utility.

Referring to Figure 20-27 — select the listed EmployeeIDs, copy them, and paste them into your text editor. You'll need them to create the modified version of the create_test_data.sql script as is shown in Example 20.8.

20.8 create_test_data.sql (Mod 2)

```
1    use EmployeeTraining
2    go
3
4    insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
5    values ('E4F786EC-D8FC-472A-9E8C-4DDE307ABEC8', 'Rick', 'Warren', 'Miller', '3/13/1961', 'M')
6    go
7    insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
8    values ('25BF3A98-AB02-445E-A5F7-67B74C6A9515', 'Steve', 'Jacob', 'Bishop', '2/10/1942', 'M')
```

 © 2008 Rick Miller — All Rights Reserved

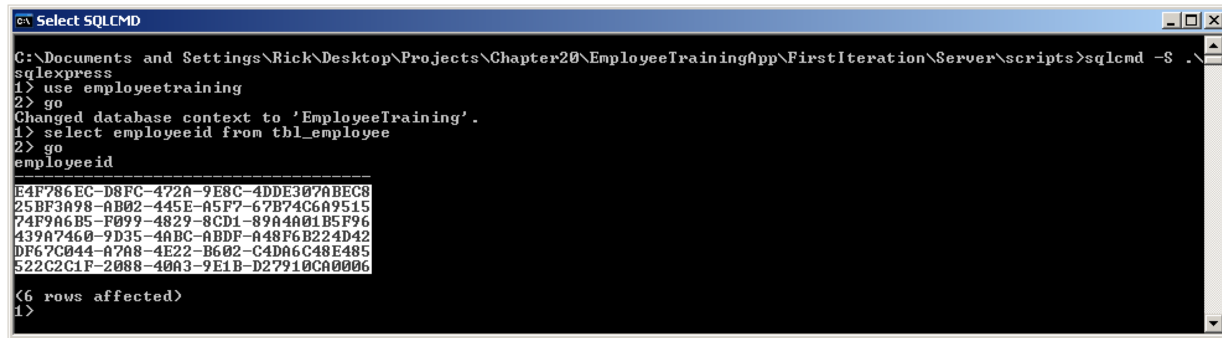


Figure 20-27: Selecting EmployeeIDs from tbl_employee

```
9    go
10   insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
11   values ('74F9A6B5-F099-4829-8CD1-89A4A01B5F96', 'Coralie', 'Sarah', 'Powell', '10/10/1974', 'F')
12   go
13   insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
14   values ('439A7460-9D35-4ABC-ABDF-A48F6B224D42', 'Kyle', 'Victor', 'Miller', '8/25/1986', 'M')
15   go
16   insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
17   values ('DF67C044-A7A8-4E22-B602-C4DA6C48E485', 'Patrick', 'Tony', 'Condemi', '4/17/1961', 'M')
18   go
19   insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
20   values ('522C2C1F-2088-40A3-9E1B-D27910CA0006', 'Dana', 'Lee', 'Condemi', '11/1/1965', 'F')
21   go
22
23
24   insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
25   values ('E4F786EC-D8FC-472A-9E8C-4DDE307ABEC8', 'Advanced Microsoft Word', 'Description text here...',
26           '11/2/2007', '11/5/2007', 'Passed')
27   go
28   insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
29   values ('E4F786EC-D8FC-472A-9E8C-4DDE307ABEC8', 'Project Management Professional',
30           'Description text here...', '6/12/2006', '6/15/2006', 'Passed')
31   go
32
33   insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
34   values ('25BF3A98-AB02-445E-A5F7-67B74C6A9515', 'Project Management Professional',
35           'Description text here...', '6/12/2006', '06/15/2006', 'Passed')
36   go
37   insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
38   values ('74F9A6B5-F099-4829-8CD1-89A4A01B5F96', 'C# Programming', 'Description text here...',
39           '1/15/2007', '5/8/2007', 'Passed')
40   go
41   insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
42   values ('439A7460-9D35-4ABC-ABDF-A48F6B224D42', 'Managing Difficult Employees',
43           'Description text here...', '1/2/2007', '1/4/2007', 'Passed')
44   go
45   insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
46   values ('439A7460-9D35-4ABC-ABDF-A48F6B224D42', 'Project Management Professional',
47           'Description text here...', '6/12/2006', '6/15/2006', 'Passed')
48   go
49   insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
50   values ('DF67C044-A7A8-4E22-B602-C4DA6C48E485', 'Squeezing Profit Margins', 'Description text here...',
51           '7/5/2004', '7/10/2004', 'Passed')
52   go
53   insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
54   values ('522C2C1F-2088-40A3-9E1B-D27910CA0006', 'Project Financial Management',
55           'Description text here...', '8/2/2007', '8/5/2007', 'Passed')
56   go
```

Referring to Example 20.8 — the modified create_test_data.sql statement does away with the newid() function and instead inserts employees into the tbl_employee table with hard-wired employee ids. These employee ids are then used to insert one or more training records into the tbl_employee_training table for each employee. Note that because of the foreign key constraint between the tbl_employee_training and tbl_employee tables, the `insert` statement checks to ensure the EmployeeID being inserted into tbl_employee_training is valid, meaning that the EmployeeID does in fact exist as a primary key in the tbl_employee table. If it were invalid the insert would fail.

To run this script be sure to first run the new create_tables.sql script to get rid of any data that may be in the tables. (Both tables should be empty at this point but if, in the future, you want to reset the test data, you'll get errors

 © 2008 Rick Miller — All Rights Reserved

if you try to run this script without first deleting the data in the tbl_employee table since the EmployeeID values are hard-wired.)

Now that we have a mix of employee and training test data loaded into the database, I can show you how to use the `select` statement to create complex queries that span multiple tables.

Selecting Data From Multiple Tables

The `select` statement can be used to perform complex database queries involving multiple tables. In this section, I show you how to use the `select` statement to *join* the tbl_employee and tbl_employee_training tables together to answer complex employee training queries.

Join Operations

Related database tables can be *joined* together to answer complex database queries. There are several different types of *join* operations but the most common one is an *inner join*, which is the default SQL Server join operation.

A join operation results in a new temporary table that contains the results of the join. A join can involve any number of related tables, or non-related tables in the case of outer joins.

Let's start by listing all the training each employee has taken and sort the results by last name. This query is shown in the following `select` statement:

```
select firstname, lastname, title
from tbl_employee, tbl_employee_training
where tbl_employee.EmployeeID = tbl_employee_training.EmployeeID
order by lastname
go
```

In this example, the `from` clause implicitly joins the tbl_employee and tbl_employee_training tables together. The `where` clause provides further filtering that limits the result set to those records in the tbl_employee table that have a matching EmployeeID entry in a tbl_employee_training record. (The term *record* is synonymous with the term *row*.) To run this query, start the SQL command utility with the following command:

```
sqlcmd -S .\sqlexpress -W
```

The `-W` switch removes the trailing spaces from each field so the query results fit on the screen.

Figure 20-28 shows the results of running this query in the SQL command utility against our freshly-loaded test data.

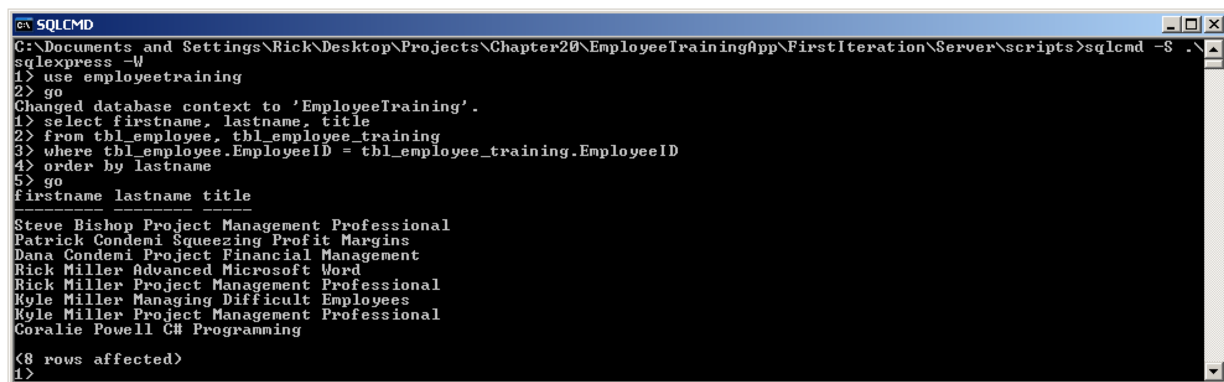


Figure 20-28: Results of Running the Previous SQL Query

Referring to Figure 20-28 — note that the number of results equals the number of training records contained in the tbl_employee_training table.

Now, suppose you wanted to find only those employees who've attended Project Management Professional training and sort the results by the employee's last name. The query for that question would look like this:

```
select firstname, lastname
from tbl_employee, tbl_employee_training
where (tbl_employee.EmployeeID = tbl_employee_training.EmployeeID) AND
```

 © 2008 Rick Miller — All Rights Reserved

```
        (title = 'Project Management Professional')
order by lastname
go
```

In this example, the `where` clause uses the `AND` operator to provide the required record filtering. The results of running this query are shown in Figure 20-29.

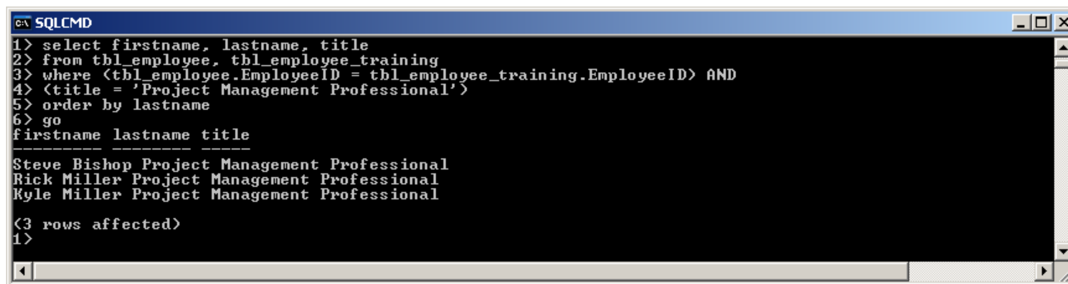


Figure 20-29: Results of Running the Previous SQL Query

Testing The Cascade Delete Constraint

If you'll return to Example 20.7 you'll see on line 37 that the foreign key constraint specifies that when a row from the tbl_employee table is deleted, all related records in the tbl_employee_training table will also be deleted. Let's test the cascade delete mechanism now by deleting the employee Rick Miller from the database. The SQL `delete` statement would look something like this:

```
delete from tbl_employee
where employeeid = 'E4F786EC-D8FC-472A-9E8C-4DDE307ABEC8'
go
```

After you execute this `delete` statement, you'll want to check to be sure the related training records were in fact deleted. You can do that with the following query:

```
select * from tbl_employee_training
```

The results of executing these two statement are shown in Figure 20-30.

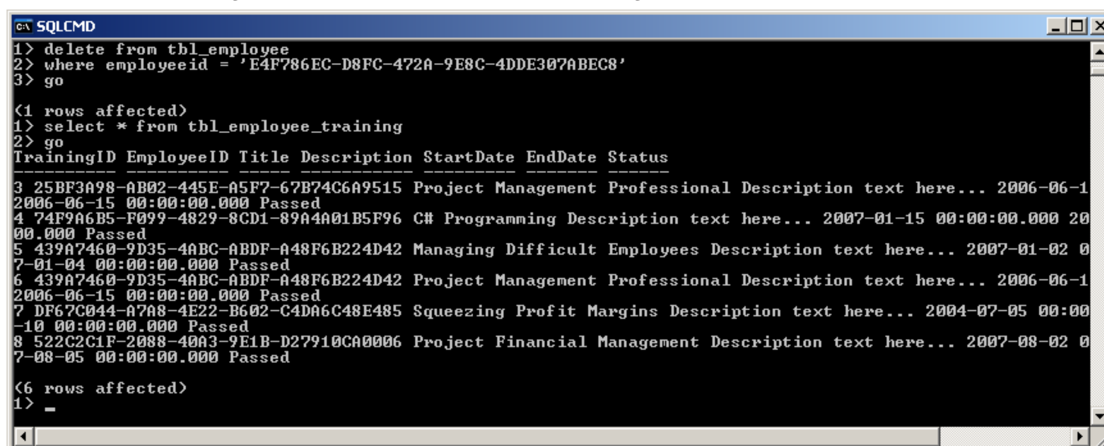


Figure 20-30: Results of Executing a Cascade Delete and Checking the Results

Quick Review

The `select` statement can be used to construct complex queries involving multiple related tables. One table is joined to another to form a temporary table. There are many different types of join operations, but the most common one is an *inner join*, which is the default join condition provided by Microsoft SQL Server.

 © 2008 Rick Miller — All Rights Reserved

Inner joins are made possible through the use of *foreign keys*. A foreign key is a column in a table that contains a value that is used as a primary key in another table. A table can be related to many other tables by including multiple foreign keys. Specify a foreign key by adding a *foreign key constraint* to a particular table using the `alter` command.

The Server Application

Now that you have a better understanding of relational databases and Structured Query Language, it's time to move on to building the employee training application. The best way to approach the design and development of a complex application is through the use of development iterations. (See Chapter 3) In this section I will step through the development of the employee training server application. As is the case with any complex development project, the best way to start is to get organized. I recommend adopting a project folder structure that mirrors the application layers or tiers.

Project Folder Organization

Figure 20-31 shows how I've arranged my server application project folders.

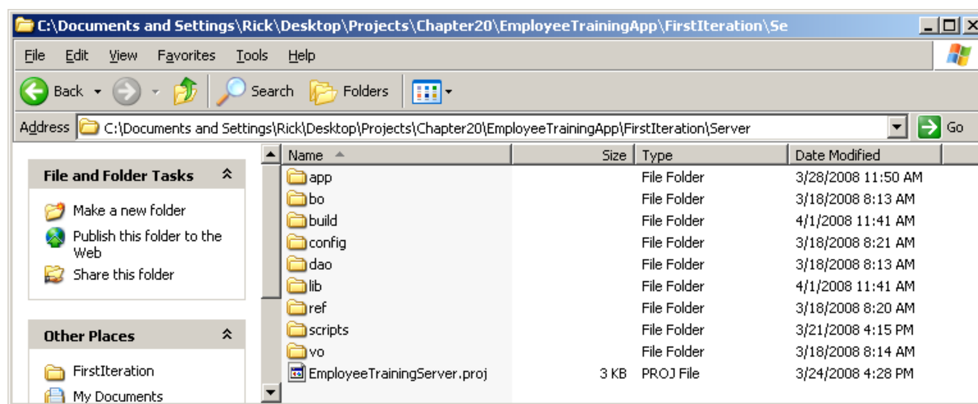


Figure 20-31: Employee Training Project Folder Arrangement

Referring to Figure 20-31 — the structure of my project folder mirrors that of the application layers contained within my application plus several more folders to hold different types of project artifacts. You'll also notice that there is an EmployeeTrainingServer.proj file at the bottom of the list. This is an MSBuild project file that is used to manage and build the project. I'll explain the use of the MSBuild project file in a moment. Table 20-2 lists and describes the purpose and contents of each of the folders shown above.

Folder Name	Contents
app	Contains source code files for the main server application, remote object interface, and remote object.
bo	Contains source code files for business objects.
build	Stores the resultant application build files. This includes dlls, the config file, and the server.exe file. Most of the files in this folder are copied from other project folders.
config	Contains the master copy of the server config file.
dao	Contains the source code files for the data access objects.

Table 20-2: Project Folder Descriptions

 © 2008 Rick Miller — All Rights Reserved

Folder Name	Contents
lib	Stores application dlls after they have been built. Other parts of the project will depend on the files stored in the lib directory.
ref	Stores dlls and other third-party libraries. These are libraries the server application depends on to build and run but are not built by the application build process.
scripts	Contains database scripts.
vo	Contains the source code files for the value objects.

Table 20-2: Project Folder Descriptions

Using Microsoft Build To Manage And Build the Project

Due to the complexity of the employee training server application, it would be difficult at best to compile the project files from the command line using only the `csc` compiler tool. The Microsoft Build tool (MSBuild) enables you to build complex projects with the help of project files. Example 20.9 gives the code for the EmployeeTraining-Server.proj file. You will find the syntax of this file somewhat confusing at first, however, keep studying it until you understand what's going on. Knowing how to use MSBuild will save you a ton of time.

20.9 EmployeeTrainingServer.proj

```
1    <Project DefaultTargets="CompileVO"
2             xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
3
4       <PropertyGroup>
5         <IncludeDebugInformation>false</IncludeDebugInformation>
6         <BuildDir>build</BuildDir>
7         <LibDir>lib</LibDir>
8         <AppDir>app</AppDir>
9         <RefDir>ref</RefDir>
10        <ConfigDir>config</ConfigDir>
11      </PropertyGroup>
12
13       <ItemGroup>
14         <DAO Include="dao\**\*.cs" />
15         <BO Include="bo\**\*.cs" />
16         <VO Include="vo\**\*.cs" />
17         <APP Include="app\**\*.cs" />
18         <LIB Include="lib\**\*.dll" />
19         <REF Include="ref\**\*.dll" />
20         <CONFIG Include="config\**\*.config" />
21         <EXE Include="app\**\*.exe" />
22       </ItemGroup>
23
24       <Target Name="MakeDirs">
25         <MakeDir Directories="$(BuildDir)" />
26         <MakeDir Directories="$(LibDir)" />
27       </Target>
28
29       <Target Name="RemoveDirs">
30         <RemoveDir Directories="$(BuildDir)" />
31         <RemoveDir Directories="$(LibDir)" />
32       </Target>
33
34       <Target Name="Clean"
35               DependsOnTargets="RemoveDirs;MakeDirs">
36       </Target>
37
38        <Target Name="CopyFiles">
39          <Copy
40            SourceFiles="@(CONFIG);@(LIB);@(REF)"
41            DestinationFolder="$(BuildDir)" />
42       </Target>
43
44       <Target Name="CompileVO"
45               Inputs="@(VO)"
46               Outputs="$(LibDir)\VOLib.dll">
47         <Csc Sources="@(VO)"
48             TargetType="library"
49             References="@(REF);@(LIB)"
50             OutputAssembly="$(LibDir)\VOLib.dll">
```


© 2008 Rick Miller — All Rights Reserved

```
51          </Csc>
52       </Target>
53
54        <Target Name="CompileDAO"
55                Inputs="@(DAO)"
56                Outputs="$(LibDir)\DAOLib.dll"
57                DependsOnTargets="CompileVO">
58           <Csc Sources="@(DAO)"
59              TargetType="library"
60              References="@(REF);@(LIB)"
61              WarningLevel="0"
62              OutputAssembly="$(LibDir)\DAOLib.dll">
63          </Csc>
64        </Target>
65
66        <Target Name="CompileBO"
67                Inputs="@(BO)"
68                Outputs="$(LibDir)\BOLib.dll"
69                DependsOnTargets="CompileDAO">
70          <Csc Sources="@(BO)"
71              TargetType="library"
72              References="@(REF);@(LIB)"
73              WarningLevel="0"
74              OutputAssembly="$(LibDir)\BOLib.dll">
75          </Csc>
76        </Target>
77
78        <Target Name="CompileApp"
79                Inputs="@(APP)"
80                Outputs="$(BuildDir)\$(MSBuildProjectName).exe"
81                DependsOnTargets="CompileDAO">
82          <Csc Sources="@(APP)"
83               TargetType="exe"
84               References="@(REF);@(LIB)"
85               OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
86          </Csc>
87        </Target>
88
89        <Target Name="CompileAll">
90          <Csc Sources="@(VO);@(DAO);@(BO);@(APP)"
91               TargetType="exe"
92               References="@(REF);@(LIB)"
93               OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
94          </Csc>
95        </Target>
96
97        <Target Name="Run"
98                DependsOnTargets="CompileApp;CopyFiles">
99          <Exec Command="$(MSBuildProjectName).exe"
100               WorkingDirectory="$(BuildDir)" />
101       </Target>
102
103  </Project>
```

Referring to Example 20.9 — the EmployeeTrainingServer.proj file contains a project specification between a pair of XML <project></project> tags. Within the project tags there appears a PropertyGroup specification, an ItemGroup specification, and several Targets.

The PropertyGroup specification appears between the <PropertyGroup></PropertyGroup> tags and defines a list of properties used within the project. Properties within the project are referenced via the $(*PropertyName*) notation. Most of the properties defined are project folder names. For example, on line 6, the <BuildDir> property is defined as the *build* directory.

The ItemGroup specification appears between the <ItemGroup></ItemGroup> tags and defines a list of project artifacts. An item within the project is referenced with the @(*ItemName*) notation. Items defined in this project include source files in various directories (.cs), library files (.dlls), config files, and executable files (.exe). For example, the DAO item defined on line 14 includes all the C# source files found in the dao directory and all its subdirectories.

The remainder of the EmployeeTrainingServer.proj file contains target definitions. A *target* is an action the MSBuild tool will perform and includes a set of one or more tasks. A target definition appears between the <Target></Target> tags. Targets can be stand-alone or they can depend on other targets. For example, the Clean target defined on line 34 depends on the RemoveDirs and MakeDirs targets. In other words, running the Clean target will also run the RemoveDirs and MakeDirs targets.

 © 2008 Rick Miller — All Rights Reserved

MSBuild projects have a default target. For example, on line 1 you see the default target for the EmployeeTrainingServer.proj is the CompileVO project.

Let's take a look at one of the more complex targets. The CompileApp target definition begins on line 78. Its inputs include all the source files in the app directory, as specified in the APP item and referenced with @(APP). Its output is an executable file written to the build directory, as specified by the BuildDir property and referenced with $(BuildDir). The CompileApp target depends on the CompileDAO target. (**Note:** This dependency will change once we move into the second iteration of the server application development.) The CompileApp target contains one compile task as is specified with the <Csc></Csc> tags. The Csc task calls the C# compiler tool and compiles all the source files found in the app directory, builds the .exe file, and writes it to the build directory. The Csc task references the libraries found in the lib and ref directories.

(**Note:** This version of the project file will change slightly as the project evolves.) As I mentioned above, the CompileApp target currently depends on the CompileDAO target. This dependency will change to the CompileBO target once we start working on the application's business objects. Also note that the CompileVO, CompileBO, and CompileDAO targets all produce dlls. These dlls are written to the lib directory.

I'll show you how to run the build using MSBuild as soon as we get some source code to compile. So, let's start on the first iteration of the employee training server application.

First Iteration

Let's see, where do we stand? The database is up and running. We have database scripts that can be used to drop and create the database, the required tables, and test data. I think a good overall objective for the first iteration of any development project is to identify, design, and code the high-risk areas. (*i.e.,* Solve the most difficult problems first.) For this project, the most difficult aspect is the DAO layer and the insertion and retrieval of an employee's data and their picture. Also, with multitier projects like this one, it's a good idea to code from the database out, meaning again that the DAO layer deserves our attention right from the start. Given this assessment, the objectives for the first development iteration are listed in Table 20-3.

Check-Off	Design Consideration	Design Decision
	DAO layer	Create a data access object (a C# class) for the employee table. Focus on the insertion and retrieval of employee data including the employee's picture. The EmployeeDAO class will need a connection to the database. This is a good use for a BaseDAO class.
	Value objects	Value objects represent entities within the application that are passed between tiers. A good place to start would be to create an EmployeeVO that contains all an employee's data. In past chapters we've already created a Person class that has most of the properties required by the EmployeeVO class. You can let the Person class serve as the base class for the EmployeeVO. For consistency we'll rename the Person class to be PersonVO.

	Enterprise Library Data Access Application Block	The Enterprise Library Data Access Application Block provides a DatabaseFactory class. You'll need to create an application configuration file that provides the required database connection. The name of the configuration file will be:
		`EmployeeTrainingServer.exe.config`
		Place this file in the project's config directory.
		You'll also need to copy and paste the following three enterprise library dlls into the project's ref directory:
		`Microsoft.Practices.EnterpriseLibrary.Common.dll`
		`Microsoft.Practices.EnterpriseLibrary.Data.dll`
		`Microsoft.Practices.ObjectBuilder.dll`

Table 20-3: Employee Training Server Application — First Iteration Design Considerations & Decisions

© 2008 Rick Miller — All Rights Reserved

Check-Off	Design Consideration	Design Decision
	Test application	You'll need to write a small application that tests the EmployeeDAO. The application should let you select an image to use for the employee's picture so it will be a GUI application. It doesn't need to be fancy as it will be thrown away. The name of the application source file will be: `EmployeeTrainingServer.cs` Create this file in the project's app directory.

Table 20-3: Employee Training Server Application — First Iteration Design Considerations & Decisions

Referring to Table 20-3 — this looks like enough work for now. Although this development cycle will yield only five source files: EmployeeTrainingServer.cs, BaseDAO.cs, EmployeeDAO.cs, PersonVO.cs, and EmployeeVO.cs, it exercises a major portion of the architecture and forces you to deal with the most complex issues you'll face during the development of this project, and that is coding up the DAO layer. I must remind you before proceeding that design decisions made early on a complex project like this one will most certainly change before the project ends. This is the natural state of affairs in software development. If the application architecture is flexible enough to be changed without too much pain then the design is sound.

Coding The EmployeeVO And EmployeeDAO

Figure 20-32 gives the UML diagram for the EmployeeVO and EmployeeDAO classes.

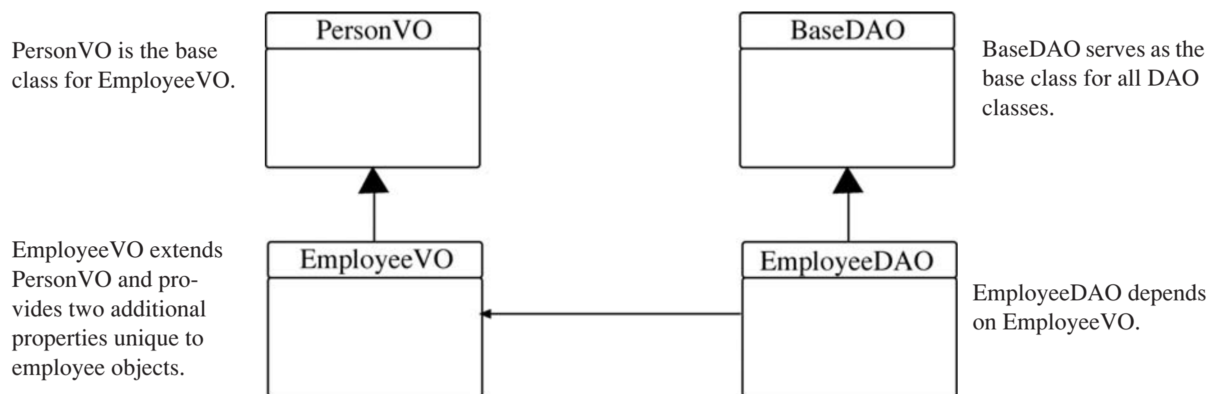


Figure 20-32: EmployeeVO and EmployeeDAO Class Diagram

Referring to Figure 20-32 — the EmployeeDAO extends the BaseDAO class and has a dependency association on the EmployeeVO class. The EmployeeVO class extends PersonVO. Since the EmployeeDAO depends on the EmployeeVO class, you must code it first. Examples 20.10 and 20.11 give the code for these classes.

20.10 PersonVO.cs

```
1    using System;
2
3    namespace EmployeeTraining.VO {
4      [Serializable]
5      public class PersonVO {
6
7      //enumeration
8      public enum Sex {MALE, FEMALE};
9
10     // private instance fields
11     private String   _firstName;
12     private String   _middleName;
13     private String   _lastName;
14     private Sex      _gender;
15     private DateTime _birthday;
16
17     //default constructor
18     public PersonVO(){}
19
20     public PersonVO(String firstName, String middleName, String lastName,
21                   Sex gender, DateTime birthday){
```

 © 2008 Rick Miller — All Rights Reserved

```
22        FirstName = firstName;
23        MiddleName = middleName;
24        LastName = lastName;
25        Gender = gender;
26        BirthDay = birthday;
27     }
28
29     // public properties
30     public String FirstName {
31       get { return _firstName; }
32       set { _firstName = value; }
33     }
34
35     public String MiddleName {
36       get { return _middleName; }
37       set { _middleName = value; }
38     }
39
40     public String LastName {
41       get { return _lastName; }
42       set { _lastName = value; }
43     }
44
45     public Sex Gender {
46       get { return _gender; }
47       set { _gender = value; }
48     }
49
50     public DateTime BirthDay {
51       get { return _birthday; }
52       set { _birthday = value; }
53     }
54
55     public int Age {
56        get {
57        int years = DateTime.Now.Year - _birthday.Year;
58          int adjustment = 0;
59        if((DateTime.Now.Month <= _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
60          adjustment = 1;
61        }
62        return years - adjustment;
63      }
64      }
65
66     public String FullName {
67       get { return FirstName + " " + MiddleName + " " + LastName; }
68     }
69
70     public String FullNameAndAge {
71       get { return FullName + " " + Age; }
72     }
73
74     public override String ToString(){
75       return FullName + " is a " + Gender + " who is " + Age + " years old.";
76     }
77
78     } // end PersonVO class
79   } // end namespace
```

Referring to Example 20.10 — the PersonVO class differs from the original Person class in a couple of ways. First, it belongs to the EmployeeTraining.VO namespace, as line 3 indicates. Second, I made the default constructor public since this class is being used as a base class and I want to be able to create EmployeeVO objects with the EmployeeVO's default constructor.

20.11 EmployeeVO.cs

```
1    using System;
2    using System.Drawing;
3
4    namespace EmployeeTraining.VO {
5    [Serializable]
6      public class EmployeeVO : PersonVO {
7
8      // private instance fields
9      private Guid     _employeeID;
10     private Image  _picture;
11
12     //default constructor
13     public EmployeeVO(){}
14
15     public EmployeeVO(Guid employeeid, String firstName, String middleName, String lastName,
```


© 2008 Rick Miller — All Rights Reserved

```
16                  Sex gender, DateTime birthday):base(firstName, middleName, lastName, gender, birthday){
17       EmployeeID = employeeid;
18     }
19
20     // public properties
21     public Guid EmployeeID {
22       get { return _employeeID;  }
23       set { _employeeID = value; }
24     }
25
26     public Image Picture {
27       get { return _picture;  }
28       set { _picture = value; }
29     }
30
31     public override String ToString(){
32       return (EmployeeID + " " + base.ToString());
33     }
34   } // end EmployeeVO class
35  } // end namespace
```

Referring to Example 20.11 — the EmployeeVO class is quite simple because most of the heavy lifting is done by the PersonVO class. This class adds two additional properties: EmployeeID, which is of type System.Guid (Globally Unique Identifier), and Picture, which is of type System.Drawing.Image. Note that both the PersonVO and EmployeeVO classes are tagged with the Serializable attribute.

To compile these classes with the MSBuild project file, make sure both classes are located in the project's vo directory, change to the server directory, and run the project file with the following command:

```
msbuild /target:compilevo
```

If you get compile errors, edit the files accordingly and run the build again. Eventually, your output should look similar to that shown in Figure 20-33.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server>msbuild /target:comp
ilevo
Microsoft (R) Build Engine Version 3.5.21022.8
[Microsoft .NET Framework, Version 2.0.50727.1433]
Copyright (C) Microsoft Corporation 2007. All rights reserved.

Build started 4/12/2008 9:55:37 AM.

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.29

C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server>
```

Figure 20-33: Results of Running the CompileVO Target using the MSBuild Utility

At this point you should check to ensure the build did in fact write the VOLib.dll to the lib directory. If not, check the validity of the EmployeeTrainingServer.proj file and make sure your project folder names match those of the properties defined within the project file. Then try and try again until you get this build target to work correctly.

Now, if you edit either the PersonVO or EmployeeVO source files and run the compilevo target again without running the clean target, you'll get the type conflict warnings shown in Figure 20-34.

You can safely ignore these warnings. What's happening here is that when the <Csc> task executes it references the VOLib.dll, which now resides in the lib directory. This dll contains the definition for the PersonVO. The warning message states that the compiler is using the definition found in PersonVO.cs instead, which is perfectly fine.

Now that the EmployeeVO is coded up, you can move on to the EmployeeDAO. You might want to write a short application to test the EmployeeVO but I'm skipping that step in this chapter. In a production environment, you'd use a testing framework like NUnit to write unit tests that thoroughly exercise the classes you create in your application. (Unfortunately, I don't have the space to cover the use of NUnit in this book but I recommend you explore its capabilities on your own when you have a chance.)

Example 20-12 gives the code for the BaseDAO class.

20.12 BaseDAO.cs

```
1    using System;
2    using System.Data;
3    using System.Configuration;
4
5    using Microsoft.Practices.EnterpriseLibrary.Data;
6
7    namespace EmployeeTraining.DAO {
```

 © 2008 Rick Miller — All Rights Reserved

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server>msbuild /target:comp
ilevo
Microsoft (R) Build Engine Version 3.5.21022.8
[Microsoft .NET Framework, Version 2.0.50727.1433]
Copyright (C) Microsoft Corporation 2007. All rights reserved.

Build started 4/12/2008 10:05:14 AM.
Project "C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\EmployeeTr
ainingServer.proj" on node 0 (compilevo target(s)).
vo\EmployeeVO.cs(6,29): warning CS0436: The type 'EmployeeTraining.VO.PersonVO' in 'c:\Documents and Settings\Rick\Desk
top\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\vo\PersonVO.cs' conflicts with the imported type 'Empl
oyeeTraining.VO.PersonVO' in 'c:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIterat
ion\Server\lib\VOLib.dll'. Using the one in 'c:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTraining
App\FirstIteration\Server\vo\PersonVO.cs'.
Done Building Project "C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Ser
ver\EmployeeTrainingServer.proj" (compilevo target(s)).


Build succeeded.

"C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\EmployeeTrainingSe
rver.proj" (compilevo target) (1) ->
(CompileVO target) ->
  vo\EmployeeVO.cs(6,29): warning CS0436: The type 'EmployeeTraining.VO.PersonVO' in 'c:\Documents and Settings\Rick\De
sktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server\vo\PersonVO.cs' conflicts with the imported type 'Em
ployeeTraining.VO.PersonVO' in 'c:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIter
ation\Server\lib\VOLib.dll'. Using the one in 'c:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTraini
ngApp\FirstIteration\Server\vo\PersonVO.cs'.

    1 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.29

C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\FirstIteration\Server>_
```

Figure 20-34: Build Warnings From Conflicting Type Declarations

```
8      public class BaseDAO {
9        private Database _database;
10
11       protected Database DataBase {
12         get {
13           if(_database == null){
14             try {
15               _database = DatabaseFactory.CreateDatabase();
16             }catch(ConfigurationException ce){
17               Console.WriteLine(ce);
18             }
19           }
20           return _database;
21         }
22       }
23
24       protected void CloseReader(IDataReader reader){
25         if(reader != null){
26           try{
27             reader.Close();
28           }catch(Exception e){
29             Console.WriteLine(e);
30           }
31         }
32       }
33
34     } // end BaseDAO class definition
35   } // end namespace
```

Referring to Example 20.12 — the purpose of this class is to make available to its subclasses a Database object via its DataBase property. This class implements the Singleton software design pattern. (See Chapter 24.) The DataBase property definition begins on line 11. If the _database field is null, a call is made to the DatabaseFactory.CreateDatabase() method to create the Database object. If the _database field is not null, the property simply returns the existing reference. This class also provides a CloseReader() method used by its subclasses to close the IDatabaseReader object.

Example 20-13 lists the 1st iteration implementation of the EmployeeDAO class. For this iteration I focused on the insertion and retrieval of EmployeeVO object data into the database. In the EmployeeDAO class you'll find all the SQL code required to create, read, update, and delete (CRUD) employee database records, although in this initial version of the code I've only implemented the create (*i.e.,* insert) and read (*i.e.,* get) operations.

20.13 EmployeeDAO.cs (1st Iteration)

```
1    using System;
2    using System.IO;
3    using System.Data;
4    using System.Data.Common;
5    using System.Data.Sql;
```

 © 2008 Rick Miller — All Rights Reserved

```
6    using System.Data.SqlTypes;
7    using System.Data.SqlClient;
8    using System.Collections.Generic;
9    using System.Drawing;
10   using System.Drawing.Imaging;
11   using EmployeeTraining.VO;
12
13   using Microsoft.Practices.EnterpriseLibrary.Common;
14   using Microsoft.Practices.EnterpriseLibrary.Data;
15   using Microsoft.Practices.EnterpriseLibrary.Data.Sql;
16
17   namespace EmployeeTraining.DAO {
18     public class EmployeeDAO : BaseDAO {
19
20      private bool debug = true;
21
22       //List of column identifiers used in perpared statements
23       private const String EMPLOYEE_ID = "@employee_id";
24       private const String FIRST_NAME = "@first_name";
25       private const String MIDDLE_NAME = "@middle_name";
26       private const String LAST_NAME = "@last_name";
27       private const String BIRTHDAY = "@birthday";
28       private const String GENDER = "@gender";
29       private const String PICTURE = "@picture";
30
31       private const String SELECT_ALL_COLUMNS =
32         "SELECT employeeid, firstname, middlename, lastname, birthday, gender, picture ";
33
34       private const String SELECT_ALL_EMPLOYEES =
35         SELECT_ALL_COLUMNS +
36         "FROM tbl_employee ";
37
38       private const String SELECT_EMPLOYEE_BY_EMPLOYEE_ID =
39         SELECT_ALL_EMPLOYEES +
40         "WHERE employeeid = " + EMPLOYEE_ID;
41
42
43       private const String INSERT_EMPLOYEE =
44         "INSERT INTO tbl_employee " +
45             "(EmployeeID, FirstName, MiddleName, LastName, Birthday, Gender, Picture) " +
46          "VALUES (" + EMPLOYEE_ID + ", " + FIRST_NAME + ", " + MIDDLE_NAME + ", " + LAST_NAME + ", " +
47                     BIRTHDAY + ", " + GENDER + ", " + PICTURE + ")";
48
49
50     /************************************
51        Returns a List<EmployeeVO> object
52     *************************************/
53       public List<EmployeeVO> GetAllEmployees(){
54         DbCommand command = DataBase.GetSqlStringCommand(SELECT_ALL_EMPLOYEES);
55         return this.GetEmployeeList(command);
56       }
57
58     /***********************************************
59        Returns an EmployeeVO object given a valid employeeid
60     ************************************************/
61       public EmployeeVO GetEmployee(Guid employeeid){
62         DbCommand command = DataBase.GetSqlStringCommand(SELECT_EMPLOYEE_BY_EMPLOYEE_ID);
63         DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
64         return this.GetEmployee(command);
65       }
66
67     /******************************************************
68        Inserts an employee given a fully-populated EmployeeVO object
69     *******************************************************/
70       public EmployeeVO InsertEmployee(EmployeeVO employee){
71         try{
72           employee.EmployeeID = Guid.NewGuid();
73           DbCommand command = DataBase.GetSqlStringCommand(INSERT_EMPLOYEE);
74           DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employee.EmployeeID);
75           DataBase.AddInParameter(command, FIRST_NAME, DbType.String, employee.FirstName);
76           DataBase.AddInParameter(command, MIDDLE_NAME, DbType.String, employee.MiddleName);
77           DataBase.AddInParameter(command, LAST_NAME, DbType.String, employee.LastName);
78           DataBase.AddInParameter(command, BIRTHDAY, DbType.DateTime, employee.BirthDay);
79           switch(employee.Gender){
80             case EmployeeVO.Sex.MALE: DataBase.AddInParameter(command, GENDER, DbType.String, "M");
81                  break;
82             case EmployeeVO.Sex.FEMALE: DataBase.AddInParameter(command, GENDER, DbType.String, "F");
83                  break;
84           }
85
86           if(employee.Picture != null){
```

 © 2008 Rick Miller — All Rights Reserved

```
87           if(debug){ Console.WriteLine("Inserting picture!"); }
88            MemoryStream ms = new MemoryStream();
89            employee.Picture.Save(ms, ImageFormat.Tiff);
90            byte[] byte_array = ms.ToArray();
91            if(debug){
92              for(int i=0; i<byte_array.Length; i++){
93                Console.Write(byte_array[i]);
94              }
95            } // end if debug
96           DataBase.AddInParameter(command, PICTURE, DbType.Binary, byte_array);
97           if(debug){ Console.WriteLine("Picture inserted, I think!"); }
98          }
99
100         DataBase.ExecuteNonQuery(command);
101       }catch(Exception e){
102         Console.WriteLine(e);
103       }
104       return this.GetEmployee(employee.EmployeeID);
105     }
106
107    /**********************************************************
108      Private utility method that executes the given DbCommand
109      and returns a fully-populated EmployeeVO object
110    **********************************************************/
111     private EmployeeVO GetEmployee(DbCommand command){
112       EmployeeVO empVO = null;
113       IDataReader reader = null;
114       try {
115         reader = DataBase.ExecuteReader(command);
116         if(reader.Read()){
117           empVO = this.FillInEmployeeVO(reader);
118         }
119       }catch(Exception e){
120         Console.WriteLine(e);
121       }finally {
122         base.CloseReader(reader);
123       }
124       return empVO;
125     }
126
127    /****************************************************
128      GetEmployeeList() - returns a List<EmployeeVO> object
129    ****************************************************/
130     private List<EmployeeVO> GetEmployeeList(DbCommand command){
131       IDataReader reader = null;
132       List<EmployeeVO> employee_list = new List<EmployeeVO>();
133       try{
134         reader = DataBase.ExecuteReader(command);
135         while(reader.Read()){
136           EmployeeVO empVO = this.FillInEmployeeVO(reader);
137           employee_list.Add(empVO);
138         }
139       }catch(Exception e){
140         Console.WriteLine(e);
141       }finally{
142         base.CloseReader(reader);
143       }
144       return employee_list;
145     }
146
147    /*****************************************************************
148      Private utility method that populates an EmployeeVO object from
149      data read from the IDataReader object
150    *****************************************************************/
151     private EmployeeVO FillInEmployeeVO(IDataReader reader){
152       EmployeeVO empVO = new EmployeeVO();
153       empVO.EmployeeID = reader.GetGuid(0);
154       empVO.FirstName = reader.GetString(1);
155       empVO.MiddleName = reader.GetString(2);
156       empVO.LastName = reader.GetString(3);
157       empVO.BirthDay = reader.GetDateTime(4);
158       String gender = reader.GetString(5);
159       switch(gender){
160         case "M" : empVO.Gender = EmployeeVO.Sex.MALE;
161                    break;
162         case "F" : empVO.Gender = EmployeeVO.Sex.FEMALE;
163                    break;
164       }
165       if(!reader.IsDBNull(6)){
166         int buffersize = 5000;
167         int startindex = 0;
```


© 2008 Rick Miller — All Rights Reserved

```
168          Byte[] byte_array = new Byte[buffersize];
169          MemoryStream ms = new MemoryStream();
170          long retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
171          while(retval > 0){
172            ms.Write(byte_array, 0, byte_array.Length);
173            startindex += buffersize;
174            retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
175           }
176          empVO.Picture = new Bitmap(ms);
177        }
178        return empVO;
179      }
180    } // end EmployeeDAO definition
181  } // end namespace
```

Referring to Example 20.13 — lines 23 through 29 define SQL command parameter string constants representing each column in the tbl_employee table. Note that these are not the same as verbatim strings. The difference lies in the placement of the @ symbol. This is a verbatim string:

`@"this is a verbatim string"`

This is an SQL command parameter string:

`"@this is an SQL parameter string"`

The SQL command parameter string constants are then used to create SQL query string constants, which are used later to create *prepared statements*. Let's see how this is done by tracing the execution of the InsertEmployee() method, which begins on line 70.

The InsertEmployee() method takes a populated EmployeeVO object as an argument. Since this is a new employee, the incoming EmployeeVO object lacks a valid EmployeeID, so the first thing that must be done is to make a call to the Guid.NewGuid() method to generate a valid globally unique identifier. This Guid value will become the employee's primary key.

On line 73, the BaseDAO's DataBase property (which is a Database object) is used to create a DbCommand object with a call to its GetSqlStringCommand() method. The argument to this method call is the INSERT_EMPLOYEE SQL string constant, which is defined on line 43. Refer now to line 43 to see how the SQL command parameters are used to formulate the INSERT_EMPLOYEE query string. Note the correspondence between each SQL command parameter included in the INSERT_EMPLOYEE string and lines 74 through 84 where the DataBase.AddInParameter() method is called to set the value of each SQL command parameter.

The `switch` statement beginning on line 79 checks the value of the incoming EmployeVO.Gender property and sets the GENDER command parameter to the corresponding valid one-character value required by the tbl_employee.Gender column.

The employee picture insertion code begins on line 86. If the incoming EmployeeVO.Picture property is null, I skip the insertion. This is valid because the tbl_employee.Picture column is allowed to contain null values. If the EmployeeVO.Picture property is not null then it's converted into a byte array (byte[]). To do this I save the Picture data to a MemoryStream and then call the MemoryStream's ToArray() method, which returns the required byte array. I've also included some debugging code that allows me to trace the insertion of the picture data if the class constant *debug* is true. (lines 87, 91, and 97)

When all the command parameters have been set, I execute the DbCommand by calling the DataBase.ExecuteNonQuery() method.

SQL Command Parameters And Prepared Statements: Generalized Steps

So, in a nutshell, here are the generalized steps to using SQL command parameters and prepared statements:

Step 1: Define the required SQL command parameters. There is usually a one-to-one correspondence between a command parameter and a column in the targeted database table.

Step 2: Create an SQL command string using the previously defined command parameters.

Step 3: Create a DbCommand object by calling the Database.GetSqlCommandString() method passing in as an argument the SQL command string.

Step 4: Set each command parameter value with a call to Database.AddInParameter() method.

Step 5: Execute the DbCommand with a call to ExecuteNonQuery() (or ExecuteReader() or ExecuteScalar() methods.)

 © 2008 Rick Miller — All Rights Reserved

DbType Enumeration Values And .NET Type Mapping

Refer for a moment to line 74 of the EmployeeDAO class. The Database.AddInParameter() method takes four arguments. These include a DbCommand reference, an SQL command parameter string, a DbType, and the value you want to use to set the SQL command parameter. The DbType enumeration is located in the System.Data namespace and defines a list of database types available to a .NET data provider. The type of the value you want to set the SQL command parameter to must correspond to the appropriate DbType, which must correspond to the MS SQL Server database type supported by the targeted database table column. Table 20-4 offers a mapping table between these three types and the corresponding IDataReader methods.

.NET Type	DbType	SQL Server Type	IDataReader Methods
String	AnsiString	varchar	GetString()
byte byte[]	Binary	varbinary	GetByte(), GetBytes()
byte	Byte	binary, varbinary	GetByte()
bool	Boolean	bit	GetBoolean()
decimal	Currency	money smallmoney	GetDecimal()
DateTime	Date	datetime smalldatetime	GetDateTime()
DateTime	DateTime	datetime smalldatetime	GetDateTime()
decimal	Decimal	decimal	GetDecimal()
double	Double	float	GetDouble()
Guid	Guid	uniqueidentifier	GetGuid()
short	Int16	smallint	GetInt16()
int	Int32	int	GetInt32()
long	Int64	bigint	GetInt64()
Object	Object	varbinary	GetValue()
sbyte	SByte	binary	GetBinary()
float	Single	float real	GetFloat()
String char[]	String	char, varchar, text nchar, nvarchar	GetChar(), GetChars() GetString()
DateTime	Time	datetime	GetDateTime()
ushort	UInt16		
uint	UInt32		
ulong	UInt64		

Table 20-4: .NET to DbType to SQL Server Type to IDataReader Method Mapping

© 2008 Rick Miller — All Rights Reserved

.NET Type	DbType	SQL Server Type	IDataReader Methods
	VarNumeric		
	AnsiStringFixedLength		
XMLDocument	Xml	xml	
DateTime	DateTime2		
DateTime	DateTimeOffset		

Table 20-4: .NET to DbType to SQL Server Type to IDataReader Method Mapping

Referring to Table 20-4 — note that there is not a one-to-one correspondence between all .NET, DbType, and SQL Server types.

Application Configuration File

Example 20-14 gives the configuration file for the first iteration of the Employee Training application. You can create this file with the Enterprise Library Configuration tool, which was covered earlier in the chapter.

20.14 EmployeeTrainingServer.exe.config (1st iteration version)

```
1    <configuration>
2      <configSections>
3        <section name="dataConfiguration"
4                 type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings,
5                 Microsoft.Practices.EnterpriseLibrary.Data, Version=3.1.0.0, Culture=neutral,
6                 PublicKeyToken=b03f5f7f11d50a3a" />
7      </configSections>
8      <dataConfiguration defaultDatabase="Connection String" />
9      <connectionStrings>
10       <add name="Connection String" connectionString="Data Source=(local)\SQLEXPRESS;
11                                                       Initial Catalog=EmployeeTraining;
12                                                       Integrated Security=True"
13                                                       providerName="System.Data.SqlClient" />
14     </connectionStrings>
15   </configuration>
```

Referring to Example 20-14 — this version provides the necessary database connection information required for the DatabaseFactory class. Later, I will add to this file a remoting section to configure the remote object, but for now it's fine the way it stands.

Creating Test Application

All that's left now is to write a brief test application that can be used to create and retrieve employee objects and test the DAO layer. Example 20-15 gives the code for a GUI application that provides a PictureBox and several buttons. The primary goal of this test application is to allow the selection and insertion of an employee picture. I do not particularly care about creating different employees per se, so there are no text boxes with which to enter employee data like an employee's first name, last name, etc. I instead create the same employee, Rick Miller.

20.15 EmployeeTrainingServer.cs (Throw away test code)

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4    using System.Drawing.Imaging;
5    using System.Collections.Generic;
6    using EmployeeTraining.DAO;
7    using EmployeeTraining.VO;
8
9    public class EmployeeTrainingServer : Form  {
10
11     private PictureBox _picturebox;
12     private TableLayoutPanel _tablepanel;
13     private FlowLayoutPanel _flowpanel;
14     private Button _button1;
15     private Button _button2;
16     private Button _button3;
17     private Button _button4;
18     private Button _button5;
```


© 2008 Rick Miller — All Rights Reserved

```
19      private EmployeeVO _emp_vo;
20      private List<EmployeeVO> _list;
21      private int _next_employee = 0;
22      private OpenFileDialog _dialog;
23
24      public EmployeeTrainingServer(){
25        this.InitializeComponent();
26        Application.Run(this);
27      }
28
29      private void InitializeComponent(){
30         this.SuspendLayout();
31        _tablepanel = new TableLayoutPanel();
32        _flowpanel = new FlowLayoutPanel();
33        _tablepanel.SuspendLayout();
34        _tablepanel.RowCount = 1;
35        _tablepanel.ColumnCount = 2;
36        _tablepanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
37        _tablepanel.Dock = DockStyle.Left;
38        _tablepanel.Width = 600;
39
40        _picturebox = new PictureBox();
41        _picturebox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
42
43        _button1 = new Button();
44        _button1.Text = "Create";
45        _button1.Click += this.CreateEmployee;
46        _button1.Enabled = false;
47
48        _button2 = new Button();
49        _button2.Text = "Load";
50        _button2.Click += this.LoadEmployee;
51        _button2.Enabled = false;
52
53        _button3 = new Button();
54        _button3.Text = "Find Picture";
55        _button3.Click += this.ShowOpenFileDialog;
56
57        _button4 = new Button();
58        _button4.Text = "Get All Employees";
59        _button4.AutoSize = true;
60        _button4.Click += this.GetAllEmployees;
61
62        _button5 = new Button();
63        _button5.Text = "Next";
64        _button5.Click += this.NextEmployee;
65        _button5.Enabled = false;
66
67        _tablepanel.Controls.Add(_picturebox);
68        _flowpanel.Controls.Add(_button1);
69        _flowpanel.Controls.Add(_button2);
70        _flowpanel.Controls.Add(_button3);
71        _flowpanel.Controls.Add(_button4);
72        _flowpanel.Controls.Add(_button5);
73        _tablepanel.Controls.Add(_flowpanel);
74
75        this.Controls.Add(_tablepanel);
76        this.Width = _tablepanel.Width;
77        this.Height = 300;
78        _tablepanel.ResumeLayout();
79        this.ResumeLayout();
80        _dialog = new OpenFileDialog();
81        _dialog.FileOk += this.LoadPicture;
82      }
83
84      public void ShowOpenFileDialog(Object sender, EventArgs e){
85        _dialog.ShowDialog();
86      }
87
88      public void LoadPicture(Object sender, EventArgs e){
89         String filename = _dialog.FileName;
90        _picturebox.Image = new Bitmap(filename);
91        this.AdjustPicturebox();
92        _button1.Enabled = true;
93      }
94
95      public void CreateEmployee(Object sender, EventArgs e){
96        EmployeeVO vo = new EmployeeVO();
97        vo.FirstName = "Rick";
98        vo.MiddleName = "Warren";
99        vo.LastName = "Miller";
```

 © 2008 Rick Miller — All Rights Reserved

```
100       vo.Gender = EmployeeVO.Sex.MALE;
101       vo.BirthDay = new DateTime(1961, 2, 4);
102       vo.Picture = _picturebox.Image;
103
104       EmployeeDAO dao = new EmployeeDAO();
105       _emp_vo = dao.InsertEmployee(vo);
106       _picturebox.Image = null;
107       _button2.Enabled = true;
108       _button1.Enabled = false;
109     }
110
111     public void LoadEmployee(Object sender, EventArgs e){
112       EmployeeDAO dao = new EmployeeDAO();
113       _emp_vo.Picture = null;
114       _emp_vo = dao.GetEmployee(_emp_vo.EmployeeID);
115       _picturebox.Image = _emp_vo.Picture;
116     }
117
118     public void GetAllEmployees(Object sender, EventArgs e){
119       EmployeeDAO dao = new EmployeeDAO();
120       _list = dao.GetAllEmployees();
121       foreach(EmployeeVO emp in _list){
122         Console.WriteLine(emp);
123       }
124       _button5.Enabled = true;
125     }
126
127     public void NextEmployee(Object sender, EventArgs e){
128       Console.WriteLine(_next_employee);
129       if(_next_employee >= _list.Count){
130         _next_employee = 0;
131       }
132       Console.WriteLine(_next_employee);
133       Console.WriteLine(_list[_next_employee]);
134       _picturebox.Image = _list[_next_employee++].Picture;
135       if(_picturebox.Image != null){
136         this.AdjustPicturebox();
137      }
138     }
139
140     private void AdjustPicturebox(){
141       this.SuspendLayout();
142       _tablepanel.SuspendLayout();
143       _picturebox.Width = _picturebox.Image.Width;
144       _picturebox.Height = _picturebox.Image.Height;
145       _tablepanel.Width = _picturebox.Image.Width + 300;
146       this.Width = _tablepanel.Width;
147       _tablepanel.ResumeLayout();
148       this.ResumeLayout();
149     }
150
151     public static void Main(){
152       new EmployeeTrainingServer();
153     }
154   }
```

Referring to Example 20-15 — this application displays a form that contains a TableLayoutPanel. The TableLayoutPanel contains a PictureBox and five buttons. To run this application, make sure you're in the directory that contains the EmployeeTrainingServer.proj file and enter the following MSBuild command on the command line:

```
msbuild /target:run
```

The startup window will look similar to Figure 20-35.

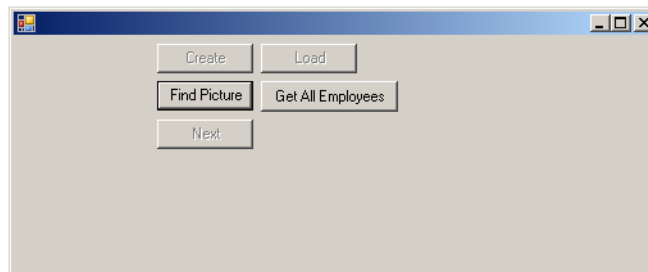


Figure 20-35: Initial State of the EmployeeTrainingServer Application Window

Referring to Figure 20-35 — you can trace the execution of the code as I discuss the use of this application. Initially, two buttons are enabled: Find Picture, and GetAllEmployees. Clicking the GetAllEmployees button calls the

 © 2008 Rick Miller — All Rights Reserved

GetAllEmployees() event handler method, which creates an EmployeeDAO object and calls its GetAllEmployees() method. The `foreach` loop on line 121 then loops through the returned list of EmployeeVO objects and prints their information to the console. Clicking the Get All Employees button also enables the Next button, which is used to step through the EmployeeVO list (_list) by calling the NextEmployee() event handler method and to display employee pictures in the picture box. Note that with an initial load of test data there will be no employee pictures, so the test application code must properly handle the possibility of the EmployeeVO.Picture property being null.

Figure 20-36 shows an employee picture loaded, and the Create button enabled. Referring to Figure 20-36 — to

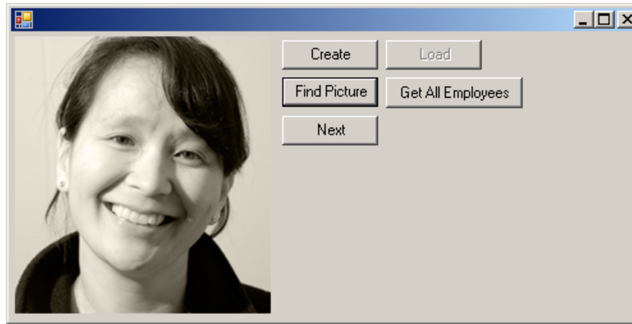


Figure 20-36: Employee Picture Loaded and Create Button Enabled

create a new employee and insert the picture into the database, click the Create button. To test the retrieval of an employee's data and picture click the Get All Employees button and then click the Next button until the picture appears in the PictureBox. Figure 20-37 shows several more employee pictures after they've been inserted and retrieved from the database.

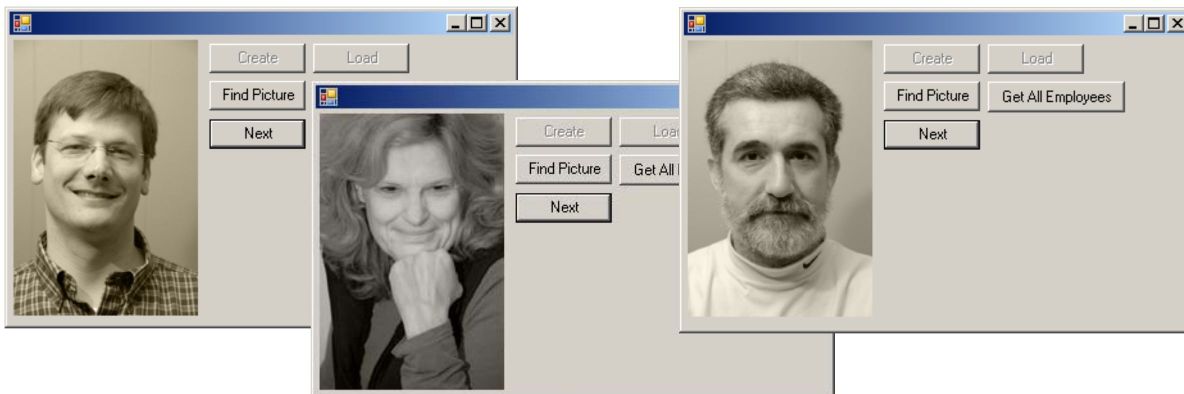


Figure 20-37: Testing with More Employee Pictures

Now, the employee ID photos I've been using are fairly small. It would be a good idea to try to load and retrieve a large image into the database. Figure 20-38 shows the results of that test.

This completes the development and testing phase of the first iteration. When you feel confident that the Empoy-eeDAO's insert and retrieval method's work fine you can move to the second iteration.

Second Iteration

A good set of objectives for the second iteration of the Employee Training application would be to finish the EmployeeDAO class by adding update and delete methods. You can also create a business object — a good name for which might be EmployeeAdminBO, and while you're at it create the TrainingDAO and TrainingVO classes. You

 © 2008 Rick Miller — All Rights Reserved

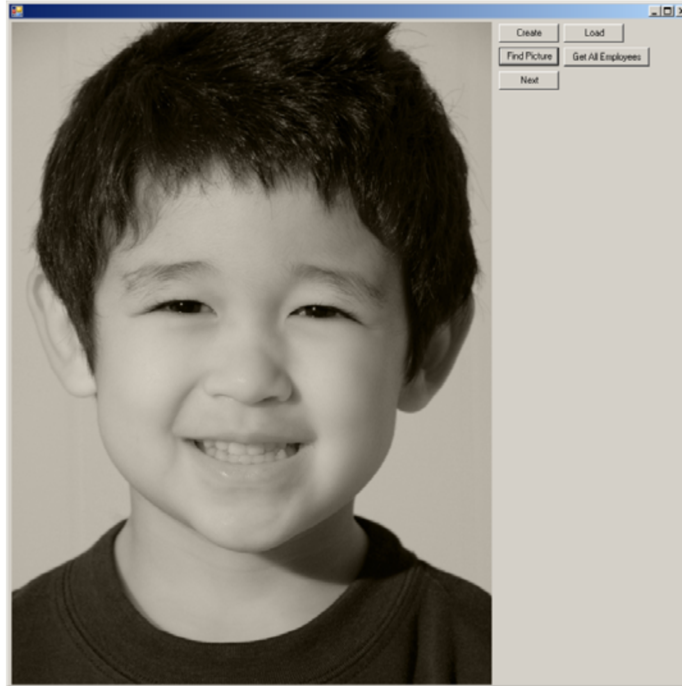


Figure 20-38: Testing the Insertion and Retrieval of a Large Image

might also want to add a few tweaks to the test application. Table 20-5 lists the design considerations and design decisions for this iteration.

Check-Off	Design Consideration	Design Decision
	DAO layer	Finish coding the EmployeeDAO. Add update and delete methods. Create the TrainingDAO class.
	Value objects	Create the TrainingVO class.
	BO layer	Create the EmployeeAdminBO class.
	Test application	Add the ability to add, update, and delete employee and employee training data. Modify the code to use the services of the EmployeeAdminBO class.
	Project file	Modify the EmployeeTrainingServer.proj file to build the contents of the bo directory.

Table 20-5: Employee Training Server Application — Second Iteration Design Considerations And Decisions

Figure 20-39 shows the UML class diagram for the TrainingDAO and TrainingVO classes.

Referring to Figure 20-39 — since the TrainingDAO class depends on the TrainingVO class, the TrainingVO class must be coded up first.

Figure 20-40 shows the UML diagram for the EmployeeAdminBO class. Referring to Figure 20-40 — the EmployeeAdminBO class has dependencies on the EmployeeVO, EmployeeDAO, TrainingVO, and TrainingDAO classes. It would be a good idea to finish coding up these four classes before starting on the EmployeeAdminBO class.

Example 20.16 gives the code for the TrainingVO class.

20.16 TrainingVO.cs

```
1    using System;
2
3    namespace EmployeeTraining.VO {
```

 © 2008 Rick Miller — All Rights Reserved

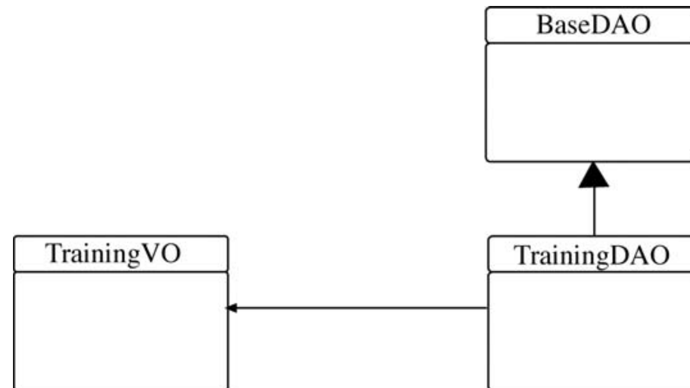


Figure 20-39: TrainingDAO and TrainingVO Class Diagram

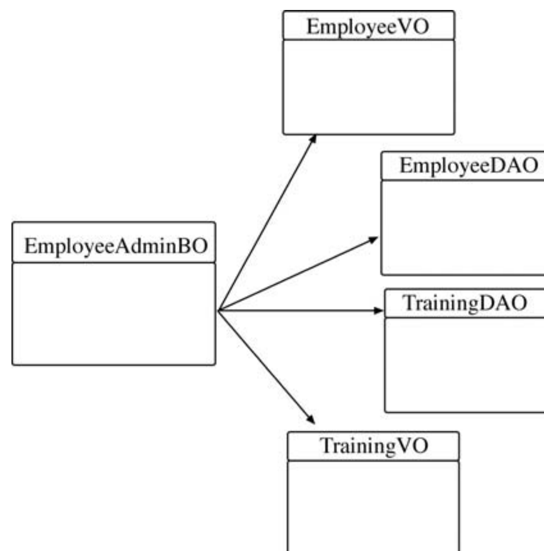


Figure 20-40: EmployeeAdminBO UML Class Diagram

```
4       [Serializable]
5      public class TrainingVO {
6
7        // Status enumeration
8        public enum TrainingStatus { Passed, Failed };
9        // Private fields
10       private int _trainingID;
11       private Guid _employeeID;
12       private String _title;
13       private String _description;
14       private DateTime _startdate;
15       private DateTime _enddate;
16       private TrainingStatus _status;
17
18       //Constructors
19       public TrainingVO(){}
20
21       public TrainingVO(int trainingID, Guid employeeID, String title, String description,
22                         DateTime startdate, DateTime enddate, TrainingStatus status){
23         TrainingID = trainingID;
24         EmployeeID = employeeID;
25         Title = title;
26         Description = description;
27         StartDate = startdate;
28         EndDate = enddate;
29         Status = status;
30       }
31
32       //Properties
33       public int TrainingID {
```

 © 2008 Rick Miller — All Rights Reserved

```
34        get { return _trainingID; }
35        set { _trainingID = value; }
36      }
37
38      public Guid EmployeeID {
39        get { return _employeeID; }
40        set { _employeeID = value; }
41      }
42
43      public String Title {
44        get { return _title; }
45        set { _title = value; }
46      }
47
48      public String Description {
49        get { return _description; }
50        set { _description = value; }
51      }
52
53      public DateTime StartDate {
54        get { return _startdate; }
55        set {_startdate = value; }
56      }
57
58      public DateTime EndDate {
59        get { return _enddate; }
60        set { _enddate = value; }
61      }
62
63      public TrainingStatus Status {
64        get { return _status; }
65        set { _status = value; }
66      }
67
68      public override String ToString(){
69        return Title + " " + Description + " " + EndDate.ToString() + " " + StartDate.ToString() +
70               " " + Status;
71      }
72
73    } // end class definition
74  } // end namespace
```

Referring to Example 20.16 — The TrainingVO class if fairly straightforward. I've added an enumeration named TrainingStatus on line 8 that contains two possible values: Passed and Failed. The TrainingStatus enumeration is used as the type for the Status property, which is defined on line 63.

Example 20-17 gives the code for the TrainingDAO class.

20.17 TrainingDAO.cs

```
1   using System;
2   using System.IO;
3   using System.Data;
4   using System.Data.Common;
5   using System.Data.Sql;
6   using System.Data.SqlTypes;
7   using System.Data.SqlClient;
8   using System.Collections.Generic;
9   using EmployeeTraining.VO;
10
11  using Microsoft.Practices.EnterpriseLibrary.Common;
12  using Microsoft.Practices.EnterpriseLibrary.Data;
13  using Microsoft.Practices.EnterpriseLibrary.Data.Sql;
14
15  namespace EmployeeTraining.DAO {
16    public class TrainingDAO : BaseDAO {
17      //List of column identifiers used in perpared statements
18      private const String TRAINING_ID = "@training_id";
19      private const String EMPLOYEE_ID = "@employee_id";
20      private const String TITLE = "@title";
21      private const String DESCRIPTION = "@description";
22      private const String STARTDATE = "@startdate";
23      private const String ENDDATE = "@enddate";
24      private const String STATUS = "@status";
25
26      // SQL statement string constants
27      private const String SELECT_ALL_COLUMNS =
28        "SELECT trainingid, employeeid, title, description, startdate, enddate, status ";
29
30      private const String SELECT_ALL_TRAINING =
31        SELECT_ALL_COLUMNS +
32        "FROM tbl_employee_training ";
```

 © 2008 Rick Miller — All Rights Reserved

```
33
34       private const String SELECT_TRAINING_BY_TRAINING_ID =
35         SELECT_ALL_TRAINING +
36         "WHERE TrainingID = " + TRAINING_ID;
37
38       private const String SELECT_TRAINING_BY_EMPLOYEE_ID =
39         SELECT_ALL_TRAINING +
40         "WHERE employeeid = " + EMPLOYEE_ID;
41
42       private const String INSERT_TRAINING =
43         "INSERT INTO tbl_employee_training " +
44             "(EmployeeID, Title, Description, StartDate, EndDate, Status) " +
45          "VALUES (" + EMPLOYEE_ID + ", " + TITLE + ", " + DESCRIPTION + ", " +
46                    STARTDATE + ", " + ENDDATE + ", " + STATUS + ") " +
47          "SELECT scope_identity()";
48
49       private const String UPDATE_TRAINING =
50         "UPDATE tbl_employee_training " +
51         "SET EmployeeID = " + EMPLOYEE_ID + ", Title = " + TITLE + ", Description = " + DESCRIPTION +
52                ", StartDate = " + STARTDATE + ", EndDate = " + ENDDATE + ", Status = " + STATUS + " " +
53         "Where TrainingID = " + TRAINING_ID;
54
55       private const String DELETE_TRAINING =
56         "DELETE FROM tbl_employee_training " +
57         "WHERE TrainingID = " + TRAINING_ID;
58
59       private const String DELETE_TRAINING_FOR_EMPLOYEEID =
60         "DELETE FROM tbl_employee_training " +
61         "WHERE EmployeeID = " + EMPLOYEE_ID;
62
63       // Public methods
64      /****************************************************************************
65        Gets a list of all training in the database.
66      ****************************************************************************/
67       public List<TrainingVO> GetAllTraining(){
68         DbCommand command = DataBase.GetSqlStringCommand(SELECT_ALL_TRAINING);
69         return this.GetTrainingList(command);
70       }
71
72      /************************************************
73        Returns a TrainingVO object given a valid trainingid
74      ************************************************/
75       public TrainingVO GetTraining(int trainingid){
76         DbCommand command = null;
77         try{
78           command = DataBase.GetSqlStringCommand(SELECT_TRAINING_BY_TRAINING_ID);
79           DataBase.AddInParameter(command, TRAINING_ID, DbType.Int32, trainingid);
80         }catch(Exception e){
81           Console.WriteLine(e);
82         }
83         return this.GetTraining(command);
84       }
85
86      /************************************************
87        Returns a List<TrainingVO> object given a valid employeeid
88      ************************************************/
89       public List<TrainingVO> GetTrainingForEmployee(Guid employeeid){
90         DbCommand command = null;
91         try{
92           command = DataBase.GetSqlStringCommand(SELECT_TRAINING_BY_EMPLOYEE_ID);
93           DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
94         }catch(Exception e){
95           Console.WriteLine(e);
96         }
97         return this.GetTrainingList(command);
98       }
99
100     /******************************************************************
101       Inserts a row into tbl_employee_training given populated TrainingVO object.
102       Returns fully-populated TrainingVO object, including primary key.
103     ******************************************************************/
104      public TrainingVO InsertTraining(TrainingVO trainingVO){
105        int trainingID = 0;
106        try{
107         DbCommand command = DataBase.GetSqlStringCommand(INSERT_TRAINING);
108         DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, trainingVO.EmployeeID);
109         DataBase.AddInParameter(command, TITLE, DbType.String, trainingVO.Title);
110         DataBase.AddInParameter(command, DESCRIPTION, DbType.String, trainingVO.Description);
111         DataBase.AddInParameter(command, STARTDATE, DbType.DateTime, trainingVO.StartDate);
112         DataBase.AddInParameter(command, ENDDATE, DbType.DateTime, trainingVO.EndDate);
113         switch(trainingVO.Status){
```


© 2008 Rick Miller — All Rights Reserved

```
114         case TrainingVO.TrainingStatus.Passed :
115              DataBase.AddInParameter(command, STATUS, DbType.String, "Passed");
116              break;
117         case TrainingVO.TrainingStatus.Failed :
118              DataBase.AddInParameter(command, STATUS, DbType.String, "Failed");
119              break;
120       }
121       trainingID = Convert.ToInt32(DataBase.ExecuteScalar(command));
122      }catch(Exception e){
123        Console.WriteLine(e);
124      }
125      return this.GetTraining(trainingID);
126    }
127
128   /**************************************************************************
129     Updates a row in the tbl_employee_training table given a populated TrainingVO object.
130   **************************************************************************/
131    public TrainingVO UpdateTraining(TrainingVO trainingVO){
132      try{
133       DbCommand command = DataBase.GetSqlStringCommand(UPDATE_TRAINING);
134       DataBase.AddInParameter(command, TRAINING_ID, DbType.Int32, trainingVO.TrainingID);
135       DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, trainingVO.EmployeeID);
136       DataBase.AddInParameter(command, TITLE, DbType.String, trainingVO.Title);
137       DataBase.AddInParameter(command, DESCRIPTION, DbType.String, trainingVO.Description);
138       DataBase.AddInParameter(command, STARTDATE, DbType.DateTime, trainingVO.StartDate);
139       DataBase.AddInParameter(command, ENDDATE, DbType.DateTime, trainingVO.EndDate);
140       switch(trainingVO.Status){
141         case TrainingVO.TrainingStatus.Passed :
142              DataBase.AddInParameter(command, STATUS, DbType.String, "Passed");
143              break;
144         case TrainingVO.TrainingStatus.Failed :
145              DataBase.AddInParameter(command, STATUS, DbType.String, "Failed");
146              break;
147       }
148       DataBase.ExecuteNonQuery(command);
149      }catch(Exception e){
150        Console.WriteLine(e);
151      }
152      return this.GetTraining(trainingVO.TrainingID);
153    }
154
155   /**************************************************************************
156     Deletes a row from the tbl_employee_training table for the given a training id.
157   **************************************************************************/
158    public void DeleteTraining(int trainingid){
159      try {
160        DbCommand command = DataBase.GetSqlStringCommand(DELETE_TRAINING);
161        DataBase.AddInParameter(command, TRAINING_ID, DbType.Int32, trainingid);
162        DataBase.ExecuteNonQuery(command);
163      }catch(Exception e){
164        Console.WriteLine(e);
165      }
166    }
167
168   /**************************************************************************
169     Deletes all training associated with given employee id.
170   **************************************************************************/
171    public void DeleteTrainingForEmployeeID(Guid employeeid){
172      try {
173        DbCommand command = DataBase.GetSqlStringCommand(DELETE_TRAINING_FOR_EMPLOYEEID);
174        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
175        DataBase.ExecuteNonQuery(command);
176      }catch(Exception e){
177        Console.WriteLine(e);
178      }
179    }
180
181   /*************************************************
182     Private utility method that executes the given DbCommand
183     and returns a fully-populated TrainingVO object
184   *************************************************/
185    private TrainingVO GetTraining(DbCommand command){
186      TrainingVO trainingVO = null;
187      IDataReader reader = null;
188      try {
189        reader = DataBase.ExecuteReader(command);
190        if(reader.Read()){
191          trainingVO = this.FillInTrainingVO(reader);
192        }
193      }catch(Exception e){
194        Console.WriteLine(e);
```

 © 2008 Rick Miller — All Rights Reserved

```
195        }finally {
196          base.CloseReader(reader);
197        }
198        return trainingVO;
199      }
200
201    /************************************************************
202       Private utility method that gets a list of TrainingVOs given a DbCommand
203    ***************************************************************/
204      private List<TrainingVO> GetTrainingList(DbCommand command){
205        IDataReader reader = null;
206        List<TrainingVO> training_list = new List<TrainingVO>();
207        try{
208          reader = DataBase.ExecuteReader(command);
209          while(reader.Read()){
210            TrainingVO trainingVO = this.FillInTrainingVO(reader);
211            training_list.Add(trainingVO);
212          }
213        }catch(Exception e){
214          Console.WriteLine(e);
215        }finally{
216          base.CloseReader(reader);
217        }
218        return training_list;
219      }
220
221    /********************************************************************
222       Private utility method that fills in a TrainingVO
223    *******************************************************************/
224      private TrainingVO FillInTrainingVO(IDataReader reader){
225        TrainingVO trainingVO = new TrainingVO();
226        trainingVO.TrainingID = reader.GetInt32(0);
227        trainingVO.EmployeeID = reader.GetGuid(1);
228        trainingVO.Title = reader.GetString(2);
229        trainingVO.Description = reader.GetString(3);
230        trainingVO.StartDate = reader.GetDateTime(4);
231        trainingVO.EndDate = reader.GetDateTime(5);
232        String status = reader.GetString(6);
233        switch(status){
234         case "Passed" : trainingVO.Status = TrainingVO.TrainingStatus.Passed;
235                         break;
236         case "Failed" : trainingVO.Status = TrainingVO.TrainingStatus.Failed;
237                         break;
238        }
239        return trainingVO;
240      }
241
242    } // end class definition
243  } // end namespace
```

Referring to Example 20.17 — the TrainingDAO class inserts, queries, updates, and deletes data in the tbl_employee_training table. This class functions like the EmployeeDAO so I'll let you walk through the code on your own.

Example 20.18 gives the completed version of the EmployeeDAO class with the delete and update methods added.

20.18 EmployeeDAO.cs (Complete)

```
1    using System;
2    using System.IO;
3    using System.Data;
4    using System.Data.Common;
5    using System.Data.Sql;
6    using System.Data.SqlTypes;
7    using System.Data.SqlClient;
8    using System.Collections.Generic;
9    using System.Drawing;
10   using System.Drawing.Imaging;
11   using EmployeeTraining.VO;
12
13   using Microsoft.Practices.EnterpriseLibrary.Common;
14   using Microsoft.Practices.EnterpriseLibrary.Data;
15   using Microsoft.Practices.EnterpriseLibrary.Data.Sql;
16
17   namespace EmployeeTraining.DAO {
18     public class EmployeeDAO : BaseDAO {
19
20      private bool debug = true;
21
22       //List of column identifiers used in perpared statements
23       private const String EMPLOYEE_ID = "@employee_id";
```


© 2008 Rick Miller — All Rights Reserved

```
24        private const String FIRST_NAME = "@first_name";
25        private const String MIDDLE_NAME = "@middle_name";
26        private const String LAST_NAME = "@last_name";
27        private const String BIRTHDAY = "@birthday";
28        private const String GENDER = "@gender";
29        private const String PICTURE = "@picture";
30
31        private const String SELECT_ALL_COLUMNS =
32          "SELECT employeeid, firstname, middlename, lastname, birthday, gender, picture ";
33
34        private const String SELECT_ALL_EMPLOYEES =
35          SELECT_ALL_COLUMNS +
36          "FROM tbl_employee ";
37
38        private const String SELECT_EMPLOYEE_BY_EMPLOYEE_ID =
39          SELECT_ALL_EMPLOYEES +
40          "WHERE employeeid = " + EMPLOYEE_ID;
41
42
43        private const String INSERT_EMPLOYEE =
44          "INSERT INTO tbl_employee " +
45              "(EmployeeID, FirstName, MiddleName, LastName, Birthday, Gender, Picture) " +
46           "VALUES (" + EMPLOYEE_ID + ", " + FIRST_NAME + ", " + MIDDLE_NAME + ", " + LAST_NAME + ", " +
47                        BIRTHDAY + ", " + GENDER + ", " + PICTURE + ")";
48
49        private const String UPDATE_EMPLOYEE =
50          "UPDATE tbl_employee " +
51          "SET FirstName = " + FIRST_NAME + ", MiddleName = " + MIDDLE_NAME + ", LastName = " + LAST_NAME +
52                 ", Birthday = " + BIRTHDAY + ", Gender = " + GENDER + ", Picture = " + PICTURE + " " +
53          "WHERE EmployeeID = " + EMPLOYEE_ID;
54
55        private const String DELETE_EMPLOYEE =
56          "DELETE FROM tbl_employee " +
57          "WHERE EmployeeID = " + EMPLOYEE_ID;
58
59      /************************************
60         Returns a List<EmployeeVO> object
61      *************************************/
62        public List<EmployeeVO> GetAllEmployees(){
63          DbCommand command = DataBase.GetSqlStringCommand(SELECT_ALL_EMPLOYEES);
64          return this.GetEmployeeList(command);
65        }
66
67      /***********************************************
68         Returns an EmployeeVO object given a valid employeeid
69      *************************************************/
70        public EmployeeVO GetEmployee(Guid employeeid){
71          DbCommand command = null;
72          try{
73            command = DataBase.GetSqlStringCommand(SELECT_EMPLOYEE_BY_EMPLOYEE_ID);
74            DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
75          }catch(Exception e){
76            Console.WriteLine(e);
77          }
78          return this.GetEmployee(command);
79        }
80
81      /*******************************************************
82         Inserts an employee given a fully-populated EmployeeVO object
83      *********************************************************/
84        public EmployeeVO InsertEmployee(EmployeeVO employee){
85          try{
86            employee.EmployeeID = Guid.NewGuid();
87            DbCommand command = DataBase.GetSqlStringCommand(INSERT_EMPLOYEE);
88            DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employee.EmployeeID);
89            DataBase.AddInParameter(command, FIRST_NAME, DbType.String, employee.FirstName);
90            DataBase.AddInParameter(command, MIDDLE_NAME, DbType.String, employee.MiddleName);
91            DataBase.AddInParameter(command, LAST_NAME, DbType.String, employee.LastName);
92            DataBase.AddInParameter(command, BIRTHDAY, DbType.DateTime, employee.BirthDay);
93            switch(employee.Gender){
94              case EmployeeVO.Sex.MALE: DataBase.AddInParameter(command, GENDER, DbType.String, "M");
95                   break;
96              case EmployeeVO.Sex.FEMALE: DataBase.AddInParameter(command, GENDER, DbType.String, "F");
97                   break;
98            }
99
100           if(employee.Picture != null){
101            if(debug){ Console.WriteLine("Inserting picture!"); }
102             MemoryStream ms = new MemoryStream();
103             employee.Picture.Save(ms, ImageFormat.Tiff);
104             byte[] byte_array = ms.ToArray();
```


© 2008 Rick Miller — All Rights Reserved

```
105            if(debug){
106              for(int i=0; i<byte_array.Length; i++){
107                Console.Write(byte_array[i]);
108              }
109            } // end if debug
110           DataBase.AddInParameter(command, PICTURE, DbType.Binary, byte_array);
111           if(debug){ Console.WriteLine("Picture inserted, I think!"); }
112          }
113
114          DataBase.ExecuteNonQuery(command);
115        }catch(Exception e){
116          Console.WriteLine(e);
117        }
118        return this.GetEmployee(employee.EmployeeID);
119      }
120
121    /********************************************************
122       Updates a row in the tbl_employee table given the fully-populated
123       EmployeeVO object.
124    *********************************************************/
125      public EmployeeVO UpdateEmployee(EmployeeVO employee){
126        try {
127          DbCommand command = DataBase.GetSqlStringCommand(UPDATE_EMPLOYEE);
128          DataBase.AddInParameter(command, FIRST_NAME, DbType.String, employee.FirstName);
129          DataBase.AddInParameter(command, MIDDLE_NAME, DbType.String, employee.MiddleName);
130          DataBase.AddInParameter(command, LAST_NAME, DbType.String, employee.LastName);
131          DataBase.AddInParameter(command, BIRTHDAY, DbType.DateTime, employee.BirthDay);
132          switch(employee.Gender){
133            case EmployeeVO.Sex.MALE: DataBase.AddInParameter(command, GENDER, DbType.String, "M");
134                 break;
135            case EmployeeVO.Sex.FEMALE: DataBase.AddInParameter(command, GENDER, DbType.String, "F");
136                 break;
137          }
138          if(employee.Picture != null){
139           if(debug){ Console.WriteLine("Inserting picture!"); }
140            MemoryStream ms = new MemoryStream();
141            employee.Picture.Save(ms, ImageFormat.Tiff);
142            byte[] byte_array = ms.ToArray();
143            if(debug){
144              for(int i=0; i<byte_array.Length; i++){
145                Console.Write(byte_array[i]);
146              }
147            } // end if debug
148           DataBase.AddInParameter(command, PICTURE, DbType.Binary, byte_array);
149           if(debug){ Console.WriteLine("Picture inserted, I think!"); }
150          }
151          DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employee.EmployeeID);
152          DataBase.ExecuteNonQuery(command);
153      }catch(Exception e){
154        Console.WriteLine(e);
155      }
156      return this.GetEmployee(employee.EmployeeID);
157      }
158
159    /********************************************************
160       Deletes a row from the tbl_employee table given an employee id.
161    ********************************************************/
162     public void DeleteEmployee(Guid employeeid){
163       try{
164         DbCommand command = DataBase.GetSqlStringCommand(DELETE_EMPLOYEE);
165         DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
166         DataBase.ExecuteNonQuery(command);
167       }catch(Exception e){
168         Console.WriteLine(e);
169       }
170     }
171
172    /************************************************
173       Private utility method that executes the given DbCommand
174       and returns a fully-populated EmployeeVO object
175    ************************************************/
176      private EmployeeVO GetEmployee(DbCommand command){
177        EmployeeVO empVO = null;
178        IDataReader reader = null;
179        try {
180          reader = DataBase.ExecuteReader(command);
181          if(reader.Read()){
182            empVO = this.FillInEmployeeVO(reader);
183          }
184        }catch(Exception e){
185          Console.WriteLine(e);
```

 © 2008 Rick Miller — All Rights Reserved

```
186        }finally {
187          base.CloseReader(reader);
188        }
189        return empVO;
190      }
191
192    /****************************************************
193      GetEmployeeList() - returns a List<EmployeeVO> object
194    *****************************************************/
195      private List<EmployeeVO> GetEmployeeList(DbCommand command){
196        IDataReader reader = null;
197        List<EmployeeVO> employee_list = new List<EmployeeVO>();
198        try{
199          reader = DataBase.ExecuteReader(command);
200          while(reader.Read()){
201            EmployeeVO empVO = this.FillInEmployeeVO(reader);
202            employee_list.Add(empVO);
203          }
204        }catch(Exception e){
205          Console.WriteLine(e);
206        }finally{
207          base.CloseReader(reader);
208        }
209        return employee_list;
210      }
211
212    /********************************************************
213      Private utility method that populates an EmployeeVO object from
214      data read from the IDataReader object
215    *********************************************************/
216      private EmployeeVO FillInEmployeeVO(IDataReader reader){
217        EmployeeVO empVO = new EmployeeVO();
218        empVO.EmployeeID = reader.GetGuid(0);
219        empVO.FirstName = reader.GetString(1);
220        empVO.MiddleName = reader.GetString(2);
221        empVO.LastName = reader.GetString(3);
222        empVO.BirthDay = reader.GetDateTime(4);
223        String gender = reader.GetString(5);
224        switch(gender){
225          case "M" : empVO.Gender = EmployeeVO.Sex.MALE;
226                     break;
227          case "F" : empVO.Gender = EmployeeVO.Sex.FEMALE;
228                     break;
229        }
230        if(!reader.IsDBNull(6)){
231          int buffersize = 5000;
232          int startindex = 0;
233          Byte[] byte_array = new Byte[buffersize];
234          MemoryStream ms = new MemoryStream();
235          long retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
236          while(retval > 0){
237            ms.Write(byte_array, 0, byte_array.Length);
238            startindex += buffersize;
239            retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
240           }
241          empVO.Picture = new Bitmap(ms);
242        }
243        return empVO;
244      }
245
246    } // end EmployeeDAO definition
247  } // end namespace
```

Example 20.19 gives the code for the EmployeeAdminBO class.

20.19 EmployeeAdminBO.cs

```
1    using System;
2    using System.Collections.Generic;
3    using EmployeeTraining.VO;
4    using EmployeeTraining.DAO;
5
6    namespace EmployeeTraining.BO {
7      public class EmployeeAdminBO {
8
9      #region Employee Methods
10
11      public EmployeeVO CreateEmployee(EmployeeVO employee){
12        EmployeeDAO dao = new EmployeeDAO();
13        return dao.InsertEmployee(employee);
14      }
15
16      public EmployeeVO GetEmployee(Guid employeeID){
```

 © 2008 Rick Miller — All Rights Reserved

```
17          EmployeeDAO dao = new EmployeeDAO();
18          return dao.GetEmployee(employeeID);
19        }
20
21        public List<EmployeeVO> GetAllEmployees(){
22          EmployeeDAO dao = new EmployeeDAO();
23          return dao.GetAllEmployees();
24        }
25
26        public EmployeeVO UpdateEmployee(EmployeeVO employee){
27          EmployeeDAO dao = new EmployeeDAO();
28          return dao.UpdateEmployee(employee);
29        }
30
31        public void DeleteEmployee(Guid employeeID){
32          EmployeeDAO dao = new EmployeeDAO();
33          dao.DeleteEmployee(employeeID);
34        }
35        #endregion Employee Methods
36
37        #region Training Methods
38        public TrainingVO CreateTraining(TrainingVO training){
39          TrainingDAO dao = new TrainingDAO();
40          return dao.InsertTraining(training);
41        }
42
43        public TrainingVO GetTraining(int trainingID){
44          TrainingDAO dao = new TrainingDAO();
45          return dao.GetTraining(trainingID);
46        }
47
48        public List<TrainingVO> GetTrainingForEmployee(Guid employeeID){
49          TrainingDAO dao = new TrainingDAO();
50          return dao.GetTrainingForEmployee(employeeID);
51        }
52
53        public TrainingVO UpdateTraining(TrainingVO training){
54          TrainingDAO dao = new TrainingDAO();
55          return dao.UpdateTraining(training);
56        }
57
58        public void DeleteTrainingForEmployee(EmployeeVO employee){
59          TrainingDAO dao = new TrainingDAO();
60          dao.DeleteTrainingForEmployeeID(employee.EmployeeID);
61        }
62
63        public void DeleteTraining(int trainingID){
64          TrainingDAO dao = new TrainingDAO();
65          dao.DeleteTraining(trainingID);
66        }
67        #endregion Training Methods
68
69      } // End class definition
70    } // End namespace
```

Referring to Example 20.19 — the EmployeeAdminBO provides methods to create, query, update, and delete employee and employee training data. The methods that deal with employee data have been grouped in the Employee Methods region by using the `#region` and `#endregion` directives. Regions allow you to collapse and expand sections of code when using Visual Studio or a compatible text editor like Notepad++. Figure 20-41 shows how code regions look when they are collapsed in Notepad++.

Referring again to Example 20.19 — in this example, all the methods are short because they are simply pass-through methods to the appropriate DAO. For example, the CreateEmployee() method defined on line 11 creates an instance of the EmployeeDAO class and calls its InsertEmployee() method to insert the EmployeeVO object's data into the tbl_employee table. In a more real world example, the EmployeeAdminBo would be used to implement and enforce more elaborate business rules. For example, if only certain types of users were allowed to create, update, or delete employee data, then those corresponding methods would perform the requisite checks to validate the user's credentials before allowing the insert, update, or delete operations via the DAO to occur.

OK, before you can compile the EmployeeAdminBO class you must make a change to the MSBuild project file. Example 20.20 shows the updated EmployeeTrainingServer.proj file.

20.20 EmployeeTrainingServer.proj (Mod 1)

```
1    <Project DefaultTargets="CompileApp"
2             xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
3
4
```

 © 2008 Rick Miller — All Rights Reserved

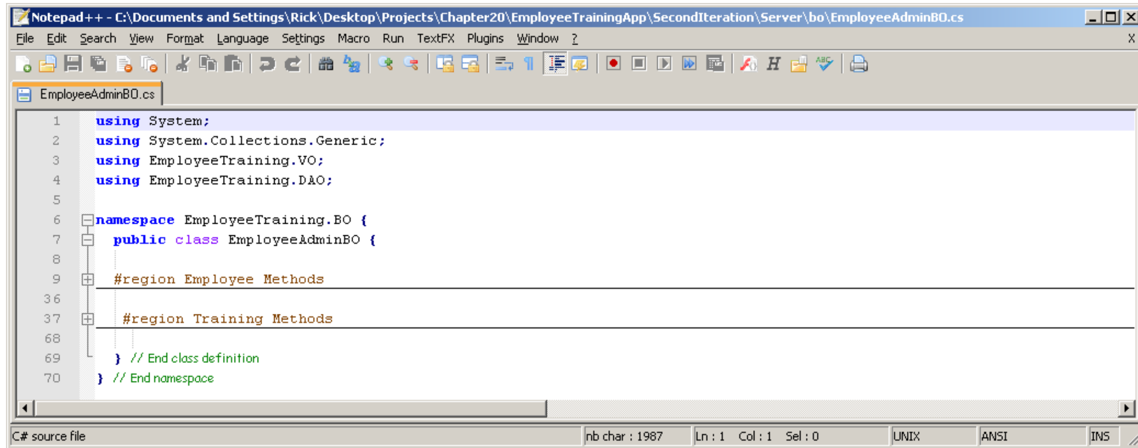


Figure 20-41: Collapsed Code Regions in Notepad++

```
5        <PropertyGroup>
6          <IncludeDebugInformation>false</IncludeDebugInformation>
7          <BuildDir>build</BuildDir>
8          <LibDir>lib</LibDir>
9          <AppDir>app</AppDir>
10         <RefDir>ref</RefDir>
11         <ConfigDir>config</ConfigDir>
12       </PropertyGroup>
13
14        <ItemGroup>
15          <DAO Include="dao\**\*.cs" />
16          <BO Include="bo\**\*.cs" />
17          <VO Include="vo\**\*.cs" />
18          <APP Include="app\**\*.cs" />
19          <LIB Include="lib\**\*.dll" />
20          <REF Include="ref\**\*.dll" />
21          <CONFIG Include="config\**\*.config" />
22          <EXE Include="app\**\*.exe" />
23        </ItemGroup>
24
25        <Target Name="MakeDirs">
26          <MakeDir Directories="$(BuildDir)" />
27          <MakeDir Directories="$(LibDir)" />
28        </Target>
29
30        <Target Name="RemoveDirs">
31          <RemoveDir Directories="$(BuildDir)" />
32          <RemoveDir Directories="$(LibDir)" />
33        </Target>
34
35        <Target Name="Clean"
36                DependsOnTargets="RemoveDirs;MakeDirs">
37        </Target>
38
39         <Target Name="CopyFiles">
40           <Copy
41             SourceFiles="@(CONFIG);@(LIB);@(REF)"
42             DestinationFolder="$(BuildDir)" />
43        </Target>
44
45        <Target Name="CompileVO"
46                Inputs="@(VO)"
47                Outputs="$(LibDir)\VOLib.dll">
48          <Csc Sources="@(VO)"
49              TargetType="library"
50              References="@(REF);@(LIB)"
51              OutputAssembly="$(LibDir)\VOLib.dll">
52          </Csc>
53       </Target>
54
55        <Target Name="CompileDAO"
56                Inputs="@(DAO)"
57                Outputs="$(LibDir)\DAOLib.dll"
58                DependsOnTargets="CompileVO">
59           <Csc Sources="@(DAO)"
60              TargetType="library"
```

 © 2008 Rick Miller — All Rights Reserved

```
61              References="@(REF);@(LIB)"
62              WarningLevel="0"
63              OutputAssembly="$(LibDir)\DAOLib.dll">
64          </Csc>
65        </Target>
66
67        <Target Name="CompileBO"
68                Inputs="@(BO)"
69                Outputs="$(LibDir)\BOLib.dll"
70                DependsOnTargets="CompileDAO">
71          <Csc Sources="@(BO)"
72              TargetType="library"
73              References="@(REF);@(LIB)"
74              WarningLevel="0"
75              OutputAssembly="$(LibDir)\BOLib.dll">
76          </Csc>
77        </Target>
78
79        <Target Name="CompileApp"
80                Inputs="@(APP)"
81                Outputs="$(BuildDir)\$(MSBuildProjectName).exe"
82                DependsOnTargets="CompileBO">
83          <Csc Sources="@(APP)"
84               TargetType="exe"
85               References="@(REF);@(LIB)"
86               OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
87          </Csc>
88        </Target>
89
90        <Target Name="CompileAll">
91          <Csc Sources="@(VO);@(DAO);@(BO);@(APP)"
92               TargetType="exe"
93               References="@(REF);@(LIB)"
94               OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
95          </Csc>
96        </Target>
97
98        <Target Name="Run"
99                DependsOnTargets="CompileApp;CopyFiles">
100         <Exec Command="$(MSBuildProjectName).exe"
101               WorkingDirectory="$(BuildDir)" />
102       </Target>
103   </Project>
```

Referring to Example 20.20 — the only changes made to the file were to the DefaultTargets on line 1, which is now set to CompileApp, and to the DependOnTargets in the CompileApp target, which is now set to CompileBO.

Testing The Code - Second Iteration

To completely test the code developed thus far requires major enhancements to the test application. Figure 20-42 shows the modified user interface of the EmployeeTrainingServer applicaton.

Referring to Figure 20-42 — the test application has been enhanced to allow the creation of employee training records as well as the ability to update and delete both employee and training data. The code for this version of the test application is given in Example 20.21.

20.21 EmployeeTrainingServer.cs (Test Application 2nd Iteration)

```
1    using System;
2    using System.Text;
3    using System.Windows.Forms;
4    using System.Drawing;
5    using System.Drawing.Imaging;
6    using System.Collections.Generic;
7    using EmployeeTraining.BO;
8    using EmployeeTraining.VO;
9
10   public class EmployeeTrainingServer : Form  {
11
12     private PictureBox _picturebox;
13     private TableLayoutPanel _tablepanel;
14
15     private FlowLayoutPanel _flowpanel;
16     private Button _create_employee_botton;
17     private Button _find_picture_button;
18     private Button _get_all_employees_button;
19     private Button _next_employee_button;
20     private Button _add_training_button;
21     private Button _update_employee_button;
```

 © 2008 Rick Miller — All Rights Reserved

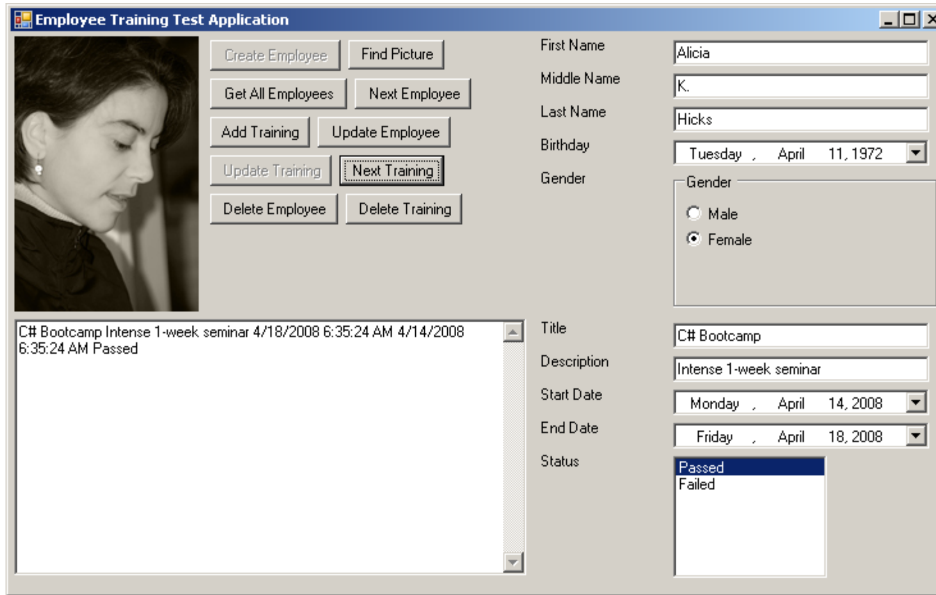


Figure 20-42: Modified Test Application

```
22      private Button _update_training_button;
23      private Button _next_training_button;
24      private Button _delete_employee_button;
25      private Button _delete_training_button;
26
27      private TableLayoutPanel _employee_info_entry_panel;
28      private Label _fname_label;
29      private Label _mname_label;
30      private Label _lname_label;
31      private Label _bday_label;
32      private Label _gender_label;
33      private TextBox _fname_textbox;
34      private TextBox _mname_textbox;
35      private TextBox _lname_textbox;
36      private DateTimePicker _bday_picker;
37      private GroupBox _gender_groupbox;
38      private RadioButton _male_button;
39      private RadioButton _female_button;
40
41
42      private const int TABLE_PANEL_ROW_COUNT = 2;
43      private const int TABLE_PANEL_COLUMN_COUNT = 3;
44      private const int TABLE_PANEL_HEIGHT = 600;
45      private const int TABLE_PANEL_WIDTH = 600;
46      private const int EMPLOYEE_INFO_PANEL_HEIGHT = 200;
47      private const int EMPLOYEE_INFO_PANEL_WIDTH= 200;
48      private const int EMPLOYEE_INFO_PANEL_ROW_COUNT = 5;
49      private const int EMPLOYEE_INFO_PANEL_COLUMN_COUNT = 2;
50      private const int TRAINING_INFO_PANEL_ROW_COUNT = 5;
51      private const int TRAINING_INFO_PANEL_COLUMN_COUNT = 2;
52      private const int TRAINING_INFO_PANEL_HEIGHT = 200;
53      private const int TRAINING_INFO_PANEL_WIDTH = 200;
54      private const int TEXTBOX_WIDTH = 200;
55      private const int SMALL_PADDING = 100;
56      private const int LARGE_PADDING = 150;
57      private const int TRAINING_TEXTBOX_WIDTH = 400;
58      private const int TRAINING_TEXTBOX_HEIGHT = 200;
59      private const int PICTUREBOX_WIDTH = 150;
60      private const int PICTUREBOX_HEIGHT = 150;
61      private const int GROUPBOX_WIDTH = 200;
62      private const int GROUPBOX_HEIGHT = 125;
63
64      private TableLayoutPanel _training_info_entry_panel;
65      private Label _title_label;
66      private Label _description_label;
67      private Label _startdate_label;
68      private Label _enddate_label;
69      private Label _status_label;
70      private TextBox _title_textbox;
71      private TextBox _description_textbox;
```

 © 2008 Rick Miller — All Rights Reserved

```
72      private DateTimePicker _startdate_picker;
73      private DateTimePicker _enddate_picker;
74      private ListBox _status_listbox;
75
76      private TextBox _training_textbox;
77
78      private EmployeeVO _emp_vo;
79      private List<EmployeeVO> _employee_list;
80      private List<TrainingVO> _training_list;
81      private int _next_employee = 0;
82      private int _next_training = 0;
83      private OpenFileDialog _dialog;
84
85      public EmployeeTrainingServer(){
86        this.InitializeComponent();
87        Application.Run(this);
88      }
89
90      private void InitializeComponent(){
91         this.SuspendLayout();
92        _tablepanel = new TableLayoutPanel();
93        _flowpanel = new FlowLayoutPanel();
94        _flowpanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
95        _tablepanel.SuspendLayout();
96        _tablepanel.RowCount = TABLE_PANEL_ROW_COUNT;
97        _tablepanel.ColumnCount = TABLE_PANEL_COLUMN_COUNT;
98        _tablepanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
99        _tablepanel.Dock = DockStyle.Top;
100
101       _picturebox = new PictureBox();
102       _picturebox.Height = PICTUREBOX_WIDTH;
103       _picturebox.Width = PICTUREBOX_HEIGHT;
104       _picturebox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
105
106       _create_employee_botton = new Button();
107       _create_employee_botton.Text = "Create Employee";
108       _create_employee_botton.AutoSize = true;
109       _create_employee_botton.Click += this.CreateEmployee;
110       _create_employee_botton.Enabled = false;
111
112       _find_picture_button = new Button();
113       _find_picture_button.Text = "Find Picture";
114       _find_picture_button.Click += this.ShowOpenFileDialog;
115
116       _get_all_employees_button = new Button();
117       _get_all_employees_button.Text = "Get All Employees";
118       _get_all_employees_button.AutoSize = true;
119       _get_all_employees_button.Click += this.GetAllEmployees;
120
121       _next_employee_button = new Button();
122       _next_employee_button.Text = "Next Employee";
123       _next_employee_button.AutoSize = true;
124       _next_employee_button.Click += this.NextEmployee;
125       _next_employee_button.Enabled = false;
126
127       _add_training_button = new Button();
128       _add_training_button.Text = "Add Training";
129       _add_training_button.AutoSize = true;
130       _add_training_button.Click += this.AddTraining;
131       _add_training_button.Enabled = false;
132
133       _update_employee_button = new Button();
134       _update_employee_button.Text = "Update Employee";
135       _update_employee_button.AutoSize = true;
136       _update_employee_button.Click += this.UpdateEmployee;
137       _update_employee_button.Enabled = false;
138
139       _update_training_button = new Button();
140       _update_training_button.Text = "Update Training";
141       _update_training_button.AutoSize = true;
142       _update_training_button.Click += this.UpdateTraining;
143       _update_training_button.Enabled = false;
144
145       _next_training_button = new Button();
146       _next_training_button.Text = "Next Training";
147       _next_training_button.AutoSize = true;
148       _next_training_button.Click += this.NextTraining;
149       _next_training_button.Enabled = false;
150
151       _delete_employee_button = new Button();
152       _delete_employee_button.Text = "Delete Employee";
```

 © 2008 Rick Miller — All Rights Reserved

```
153      _delete_employee_button.AutoSize = true;
154      _delete_employee_button.Click += this.DeleteEmployee;
155      _delete_employee_button.Enabled = false;
156
157      _delete_training_button = new Button();
158      _delete_training_button.Text = "Delete Training";
159      _delete_training_button.AutoSize = true;
160      _delete_training_button.Click += this.DeleteTraining;
161      _delete_training_button.Enabled = false;
162
163      _tablepanel.Controls.Add(_picturebox);
164      _flowpanel.Controls.Add(_create_employee_botton);
165      _flowpanel.Controls.Add(_find_picture_button);
166      _flowpanel.Controls.Add(_get_all_employees_button);
167      _flowpanel.Controls.Add(_next_employee_button);
168      _flowpanel.Controls.Add(_add_training_button);
169      _flowpanel.Controls.Add(_update_employee_button);
170      _flowpanel.Controls.Add(_update_training_button);
171      _flowpanel.Controls.Add(_next_training_button);
172      _flowpanel.Controls.Add(_delete_employee_button);
173      _flowpanel.Controls.Add(_delete_training_button);
174
175      _tablepanel.Controls.Add(_flowpanel);
176
177      _employee_info_entry_panel = new TableLayoutPanel();
178      _employee_info_entry_panel.SuspendLayout();
179      _employee_info_entry_panel.Height = EMPLOYEE_INFO_PANEL_HEIGHT;
180      _employee_info_entry_panel.Width = EMPLOYEE_INFO_PANEL_WIDTH;
181      _employee_info_entry_panel.RowCount = EMPLOYEE_INFO_PANEL_ROW_COUNT;
182      _employee_info_entry_panel.ColumnCount = EMPLOYEE_INFO_PANEL_COLUMN_COUNT;
183      _fname_label = new Label();
184      _fname_label.Text = "First Name";
185      _mname_label = new Label();
186      _mname_label.Text = "Middle Name";
187      _lname_label = new Label();
188      _lname_label.Text = "Last Name";
189      _bday_label = new Label();
190      _bday_label.Text = "Birthday";
191      _gender_label = new Label();
192      _gender_label.Text = "Gender";
193      _fname_textbox = new TextBox();
194      _fname_textbox.Width = TEXTBOX_WIDTH;
195      _mname_textbox = new TextBox();
196      _mname_textbox.Width = TEXTBOX_WIDTH;
197      _lname_textbox = new TextBox();
198      _lname_textbox.Width = TEXTBOX_WIDTH;
199      _bday_picker = new DateTimePicker();
200      _gender_groupbox = new GroupBox();
201      _gender_groupbox.Text = "Gender";
202      _gender_groupbox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left
203                                | AnchorStyles.Right;
204      _gender_groupbox.Height = GROUPBOX_HEIGHT;
205      _gender_groupbox.Width = GROUPBOX_WIDTH;
206
207      _male_button = new RadioButton();
208      _male_button.Text = "Male";
209      _male_button.Checked = true;
210      _male_button.Location = new Point(10, 20);
211      _female_button = new RadioButton();
212      _female_button.Text = "Female";
213      _female_button.Location = new Point(10, 40);
214      _gender_groupbox.Controls.Add(_male_button);
215      _gender_groupbox.Controls.Add(_female_button);
216      _gender_groupbox.Size = new Size(50, 50);
217      _employee_info_entry_panel.Controls.Add(_fname_label);
218      _employee_info_entry_panel.Controls.Add(_fname_textbox);
219      _employee_info_entry_panel.Controls.Add(_mname_label);
220      _employee_info_entry_panel.Controls.Add(_mname_textbox);
221      _employee_info_entry_panel.Controls.Add(_lname_label);
222      _employee_info_entry_panel.Controls.Add(_lname_textbox);
223      _employee_info_entry_panel.Controls.Add(_bday_label);
224      _employee_info_entry_panel.Controls.Add(_bday_picker);
225      _employee_info_entry_panel.Controls.Add(_gender_label);
226      _employee_info_entry_panel.Controls.Add(_gender_groupbox);
227      _employee_info_entry_panel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left
228                                          | AnchorStyles.Right;
229
230      _tablepanel.Controls.Add(_employee_info_entry_panel);
231
232     _training_info_entry_panel = new TableLayoutPanel();
233     _training_info_entry_panel.RowCount = TRAINING_INFO_PANEL_ROW_COUNT;
```

 © 2008 Rick Miller — All Rights Reserved

```
234     _training_info_entry_panel.ColumnCount = TRAINING_INFO_PANEL_COLUMN_COUNT;
235     _training_info_entry_panel.Height = TRAINING_INFO_PANEL_HEIGHT;
236     _training_info_entry_panel.Width = TRAINING_INFO_PANEL_WIDTH;
237     _title_label = new Label();
238     _title_label.Text = "Title";
239     _description_label = new Label();
240     _description_label.Text = "Description";
241     _startdate_label = new Label();
242     _startdate_label.Text = "Start Date";
243     _enddate_label = new Label();
244     _enddate_label.Text = "End Date";
245     _status_label = new Label();
246     _status_label.Text = "Status";
247     _title_textbox = new TextBox();
248     _title_textbox.Width = TEXTBOX_WIDTH;
249     _description_textbox = new TextBox();
250     _description_textbox.Width = TEXTBOX_WIDTH;
251     _startdate_picker = new DateTimePicker();
252     _enddate_picker = new DateTimePicker();
253     _status_listbox = new ListBox();
254     _status_listbox.Items.Add("Passed");
255     _status_listbox.Items.Add("Failed");
256     _status_listbox.SetSelected(0, true);
257
258     _training_info_entry_panel.Controls.Add(_title_label);
259     _training_info_entry_panel.Controls.Add(_title_textbox);
260     _training_info_entry_panel.Controls.Add(_description_label);
261     _training_info_entry_panel.Controls.Add(_description_textbox);
262     _training_info_entry_panel.Controls.Add(_startdate_label);
263     _training_info_entry_panel.Controls.Add(_startdate_picker);
264     _training_info_entry_panel.Controls.Add(_enddate_label);
265     _training_info_entry_panel.Controls.Add(_enddate_picker);
266     _training_info_entry_panel.Controls.Add(_status_label);
267     _training_info_entry_panel.Controls.Add(_status_listbox);
268     _training_info_entry_panel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left
269                                        | AnchorStyles.Right;
270
271     _tablepanel.Controls.Add(_training_info_entry_panel);
272
273     _training_textbox = new TextBox();
274     _training_textbox.Multiline = true;
275     _training_textbox.ScrollBars = ScrollBars.Vertical;
276     _training_textbox.Dock = DockStyle.Top;
277     _training_textbox.Width = TRAINING_TEXTBOX_WIDTH;
278     _training_textbox.Height = TRAINING_TEXTBOX_HEIGHT;
279     _tablepanel.Controls.Add(_training_textbox);
280     _tablepanel.SetRow(_training_textbox, 1);
281     _tablepanel.SetColumn(_training_textbox, 0);
282     _tablepanel.SetColumnSpan(_training_textbox, 2);
283
284     this.Controls.Add(_tablepanel);
285     _tablepanel.Width =  _training_textbox.Width + _employee_info_entry_panel.Width + LARGE_PADDING;
286     _tablepanel.Height = TABLE_PANEL_HEIGHT;
287     this.Width = _tablepanel.Width;
288     this.Height = _tablepanel.Height;
289     this.Text = "Employee Training Test Application";
290     _employee_info_entry_panel.ResumeLayout();
291     _tablepanel.ResumeLayout();
292     this.ResumeLayout();
293     _dialog = new OpenFileDialog();
294     _dialog.FileOk += this.LoadPicture;
295    }
296
297    public void ShowOpenFileDialog(Object sender, EventArgs e){
298      this.ResetEntryFields();
299      this.ResetTrainingTextbox();
300      _add_training_button.Enabled = false;
301      _delete_employee_button.Enabled = false;
302      _update_employee_button.Enabled = false;
303      _next_training_button.Enabled = false;
304      _dialog.ShowDialog();
305    }
306
307    public void LoadPicture(Object sender, EventArgs e){
308       String filename = _dialog.FileName;
309      _picturebox.Image = new Bitmap(filename);
310      this.AdjustAppWindowSize();
311      _create_employee_botton.Enabled = true;
312    }
313
314    public void CreateEmployee(Object sender, EventArgs e){
```


© 2008 Rick Miller — All Rights Reserved

```
315       EmployeeVO vo = new EmployeeVO();
316       vo = this.PopulateEmployeeVOFromEntryFields(vo);
317
318       EmployeeAdminBO bo = new EmployeeAdminBO();
319       _emp_vo = bo.CreateEmployee(vo);
320       _picturebox.Image = null;
321       _create_employee_botton.Enabled = false;
322       this.ResetEntryFields();
323       this.DisplayEmployeeInfo();
324       this.DisplayEmployeeTraining(bo);
325     }
326
327     public void GetAllEmployees(Object sender, EventArgs e){
328       EmployeeAdminBO bo = new EmployeeAdminBO();
329       _employee_list = bo.GetAllEmployees();
330       foreach(EmployeeVO emp in _employee_list){
331         Console.WriteLine(emp);
332       }
333       _next_employee_button.Enabled = true;
334     }
335
336     public void NextEmployee(Object sender, EventArgs e){
337       _next_employee++;
338       _next_training = 0;
339       Console.WriteLine(_next_employee);
340       if(_next_employee >= _employee_list.Count){
341         _next_employee = 0;
342       }
343       Console.WriteLine(_next_employee);
344       if(_employee_list.Count > 0){
345         Console.WriteLine(_employee_list[_next_employee]);
346         _emp_vo = _employee_list[_next_employee];
347         this.DisplayEmployeeInfo();
348         this.DisplayEmployeeTraining(new EmployeeAdminBO());
349         if(_training_list.Count > 0){
350           _update_training_button.Enabled = true;
351           _next_training_button.Enabled = true;
352         }else{
353           _update_training_button.Enabled = false;
354           _next_training_button.Enabled = false;
355           _delete_training_button.Enabled = false;
356         }
357         _delete_employee_button.Enabled = true;
358         _add_training_button.Enabled = true;
359         _update_employee_button.Enabled = true;
360       }else{
361         _delete_employee_button.Enabled = false;
362         _add_training_button.Enabled = false;
363         _update_employee_button.Enabled = false;
364       }
365       this.ResetTrainingEntryFields();
366     }
367
368     public void UpdateEmployee(Object sender, EventArgs e){
369       _emp_vo = this.PopulateEmployeeVOFromEntryFields(_emp_vo);
370       EmployeeAdminBO bo = new EmployeeAdminBO();
371       _emp_vo = bo.UpdateEmployee(_emp_vo);
372       this.ResetEntryFields();
373       this.DisplayEmployeeInfo();
374       this.DisplayEmployeeTraining(bo);
375     }
376
377     public void AddTraining(Object sender, EventArgs e){
378       TrainingVO vo = new TrainingVO();
379       vo = this.PopulateTrainingVOFromEntryFields(vo);
380       EmployeeAdminBO bo = new EmployeeAdminBO();
381       bo.CreateTraining(vo);
382       this.DisplayEmployeeTraining(bo);
383       this.ResetTrainingEntryFields();
384       _next_training_button.Enabled = true;
385     }
386
387     public void NextTraining(Object Sender, EventArgs e){
388       _next_training++;
389       if(_next_training >= _training_list.Count){
390         _next_training = 0;
391       }
392       if(_training_list.Count > 0){
393         this.DisplayTrainingInfo(_training_list[_next_training]);
394         _delete_training_button.Enabled = true;
395       }
```

 © 2008 Rick Miller — All Rights Reserved

```
396     }
397
398     public void UpdateTraining(Object Sender, EventArgs e){
399       EmployeeAdminBO bo = new EmployeeAdminBO();
400       bo.UpdateTraining(this.PopulateTrainingVOFromEntryFields(_training_list[_next_training]));
401       _training_list = bo.GetTrainingForEmployee(_emp_vo.EmployeeID);
402       this.DisplayEmployeeTraining(bo);
403     }
404
405     public void DeleteEmployee(Object sender, EventArgs e){
406       EmployeeAdminBO bo = new EmployeeAdminBO();
407       bo.DeleteEmployee(_emp_vo.EmployeeID);
408       _employee_list = bo.GetAllEmployees();
409       _next_employee = 0;
410       _emp_vo = null;
411       this.ResetEntryFields();
412       if(_employee_list.Count > 0){
413         _emp_vo = _employee_list[_next_employee];
414         this.DisplayEmployeeInfo();
415         this.DisplayEmployeeTraining(new EmployeeAdminBO());
416         if(_training_list.Count > 0){
417           _update_training_button.Enabled = true;
418           _next_training_button.Enabled = true;
419           _delete_training_button.Enabled = true;
420         }else{
421           _update_training_button.Enabled = false;
422           _next_training_button.Enabled = false;
423           _delete_training_button.Enabled = false;
424         }
425         _delete_employee_button.Enabled = true;
426       }else{
427         _delete_employee_button.Enabled = false;
428         _delete_training_button.Enabled = false;
429         _next_training_button.Enabled = false;
430         _update_training_button.Enabled = false;
431         _update_employee_button.Enabled = false;
432         _next_employee_button.Enabled = false;
433         this.ResetTrainingTextbox();
434       }
435     }
436
437     public void DeleteTraining(Object sender, EventArgs e){
438        EmployeeAdminBO bo = new EmployeeAdminBO();
439        bo.DeleteTraining(_training_list[_next_training].TrainingID);
440        this.DisplayEmployeeTraining(bo);
441        if(_training_list.Count > 0){
442           _update_training_button.Enabled = true;
443           _next_training_button.Enabled = true;
444           _delete_training_button.Enabled = true;
445         }else{
446           _update_training_button.Enabled = false;
447           _next_training_button.Enabled = false;
448           _delete_training_button.Enabled = false;
449         }
450         _next_training = 0;
451         this.ResetTrainingEntryFields();
452     }
453
454     private void AdjustAppWindowSize(){
455       this.SuspendLayout();
456       _tablepanel.SuspendLayout();
457       _employee_info_entry_panel.SuspendLayout();
458       _training_info_entry_panel.SuspendLayout();
459       _picturebox.Width = _picturebox.Image.Width;
460       _picturebox.Height = _picturebox.Image.Height;
461       _employee_info_entry_panel.Height = EMPLOYEE_INFO_PANEL_HEIGHT;
462       _employee_info_entry_panel.Width = EMPLOYEE_INFO_PANEL_WIDTH;
463       _training_info_entry_panel.Height = TRAINING_INFO_PANEL_HEIGHT;
464       _training_info_entry_panel.Width = TRAINING_INFO_PANEL_WIDTH;
465       _training_textbox.Width = TRAINING_TEXTBOX_WIDTH;
466       _training_textbox.Height = TRAINING_TEXTBOX_HEIGHT;
467       _tablepanel.Width = (_picturebox.Width + _flowpanel.Width + _employee_info_entry_panel.Width
468                           + SMALL_PADDING);
469       _tablepanel.Height = (_picturebox.Image.Height + _training_textbox.Height + SMALL_PADDING);
470       this.Width = _tablepanel.Width + SMALL_PADDING;
471       this.Height = _tablepanel.Height;
472       _training_info_entry_panel.ResumeLayout();
473       _employee_info_entry_panel.ResumeLayout();
474       _tablepanel.ResumeLayout();
475       this.ResumeLayout();
476     }
```


© 2008 Rick Miller — All Rights Reserved

```
477
478     private void DisplayEmployeeTraining(EmployeeAdminBO bo){
479       _training_list = bo.GetTrainingForEmployee(_emp_vo.EmployeeID);
480       _training_textbox.Text = String.Empty;
481       StringBuilder sb = new StringBuilder();
482       foreach(TrainingVO t in _training_list){
483         sb.Append(t.ToString() + "\r\n");
484       }
485       _training_textbox.Text = sb.ToString();
486     }
487
488     private TrainingVO.TrainingStatus StringToTrainingStatus(String s){
489       TrainingVO.TrainingStatus status = TrainingVO.TrainingStatus.Passed;
490       switch(s){
491         case "Passed" : status = TrainingVO.TrainingStatus.Passed;
492                         break;
493         case "Failed" : status = TrainingVO.TrainingStatus.Failed;
494                         break;
495       }
496       return status;
497     }
498
499     private void ResetEntryFields(){
500       _fname_textbox.Text = String.Empty;
501       _mname_textbox.Text = String.Empty;
502       _lname_textbox.Text = String.Empty;
503       _male_button.Checked = true;
504       _bday_picker.Value = DateTime.Now;
505       _picturebox.Image = null;
506       this.ResetTrainingEntryFields();
507     }
508
509     private void ResetTrainingEntryFields(){
510      _title_textbox.Text = String.Empty;
511      _description_textbox.Text = String.Empty;
512      _startdate_picker.Value = DateTime.Now;
513      _enddate_picker.Value = DateTime.Now;
514      _status_listbox.SetSelected(0, true);
515     }
516
517     public void ResetTrainingTextbox(){
518       _training_textbox.Text = String.Empty;
519     }
520
521     private void DisplayEmployeeInfo(){
522       _fname_textbox.Text = _emp_vo.FirstName;
523       _mname_textbox.Text = _emp_vo.MiddleName;
524       _lname_textbox.Text = _emp_vo.LastName;
525       switch(_emp_vo.Gender){
526         case PersonVO.Sex.MALE : _male_button.Checked = true;
527                                  break;
528         case PersonVO.Sex.FEMALE : _female_button.Checked = true;
529                                  break;
530       }
531       _bday_picker.Value = _emp_vo.BirthDay;
532       _picturebox.Image = _emp_vo.Picture;
533       if(_picturebox.Image != null){
534         this.AdjustAppWindowSize();
535      }
536     }
537
538     private PersonVO.Sex RadioButtonToSexEnum(){
539       PersonVO.Sex gender = PersonVO.Sex.MALE;
540       if(_male_button.Checked){
541         gender = PersonVO.Sex.MALE;
542       }else{
543         if(_female_button.Checked){
544           gender = PersonVO.Sex.FEMALE;
545         }
546       }
547       return gender;
548     }
549
550     private EmployeeVO PopulateEmployeeVOFromEntryFields(EmployeeVO vo){
551       vo.FirstName = _fname_textbox.Text;
552       vo.MiddleName = _mname_textbox.Text;
553       vo.LastName = _lname_textbox.Text;
554       vo.Gender = this.RadioButtonToSexEnum();
555       vo.BirthDay = _bday_picker.Value;
556       vo.Picture = _picturebox.Image;
557       return vo;
```

 © 2008 Rick Miller — All Rights Reserved

```
558    }
559
560    private TrainingVO PopulateTrainingVOFromEntryFields(TrainingVO vo){
561      vo.EmployeeID = _emp_vo.EmployeeID;
562      vo.Title = _title_textbox.Text;
563      vo.Description = _description_textbox.Text;
564      vo.StartDate = _startdate_picker.Value;
565      vo.EndDate = _enddate_picker.Value;
566      vo.Status = this.StringToTrainingStatus(_status_listbox.SelectedItem.ToString());
567      return vo;
568    }
569
570    private void DisplayTrainingInfo(TrainingVO vo){
571      _title_textbox.Text = vo.Title;
572      _description_textbox.Text = vo.Description;
573      _startdate_picker.Value = vo.StartDate;
574      _enddate_picker.Value = vo.EndDate;
575      switch(vo.Status){
576        case TrainingVO.TrainingStatus.Passed :
577                     _status_listbox.SetSelected(0, true);
578                     break;
579        case TrainingVO.TrainingStatus.Failed :
580                     _status_listbox.SetSelected(1, true);
581                     break;
582      }
583    }
584
585    public static void Main(){
586      new EmployeeTrainingServer();
587    }
588  }
```

Referring to Example 20.21 — you may be thinking, "Holy cow, you wrote 588 lines of test code?" Trust me, that's nothing. If you were using a test framework like NUnit to write unit tests for all the individual classes (EmployeeVO, TrainingVO, EmployeeDAO, TrainingDAO, and EmployeeAdminBO), you'd have written more than 588 lines of code, especially if your tests were well thought out and thorough. However, the more effort you put into good unit testing, the easier your programming life becomes, especially when you start to make changes to your code.

The drawbacks to using a GUI application like Example 20.21 to test your code is that it is not automatic. You must make sure to perform all the tasks manually, like creating employees, updating employees, deleting employees, and the same with their associated training records. But it's better than nothing.

You might also ask, "Why don't you just wait until you build the client to test the code?" That's not a good idea because you really do want to test as you go. You want to move into the client development iteration knowing the server code has been thoroughly tested.

Reality Check

Each development iteration actually comprises many subiterations. For example, the code developed during this second iteration took me about twenty-five subiterations of coding, compiling, and testing.

Third Iteration

At this point the server-side code is nearly complete. All that's left to do is to create the remote object and modify the EmployeeTrainingServer code to host the remote object. This will also require a modification to the EmployeeTrainingServer.exe.config file. I will also need to modify the MSBuild project file slightly to add several special build tasks to correctly build the remote object and the EmployeeTrainingServer application. Also, to test the remote object, I'll need to write a short remote client application. Table 20-6 lists the design considerations and design decisions for the third iteration.

Check-Off	Design Consideration	Design Decision
	Remote object interface	Create an interface for the remote object. I'll name the interface IEmployeeTraining. The interface will declare all the methods required to manage employee and training objects.

Table 20-6: Employee Training Server Application — Third Iteration Design Considerations And Decisions

 © 2008 Rick Miller — All Rights Reserved

Check-Off	Design Consideration	Design Decision
	Remote object	Create the remote object by extending MarshalByRefObject and implementing the IEmployeeTraining interface. I'll name the remote object EmployeeTrainingRemoteObject.
	EmployeeTrainingServer	Remove the GUI test code and add the code required to host the remote object.
	Configuration file	Add a remoting section.
	Client test application	Start coding the client application. Create a short test application that tests the remote server object. This will require the addition of a client configuration file. The required value object dlls will need to be copied to the client project folder. While I'm at it I'll create an MSBuild project file to help build and manage the client development process.

Table 20-6: Employee Training Server Application — Third Iteration Design Considerations And Decisions

Figure 20-43 shows the UML class diagram for the EmployeeTrainingRemoteObject class. Referring to Figure

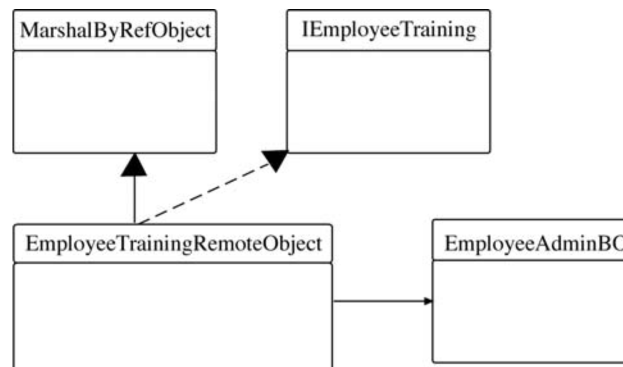


Figure 20-43: EmployeeTrainingRemoteObject UML Class Diagram

20-43 — the EmployeeTrainingRemoteObject class extends MarshalByRefObject and implements the IEmployeeTraining interface. It also uses the services of the EmployeeAdminBO class.

Example 20.22 gives the code for the IEmployeeTraining interface.

20.22 IEmployeeTraining.cs

```
1    using System;
2    using System.Collections.Generic;
3    using EmployeeTraining.VO;
4
5    public interface IEmployeeTraining {
6
7      #region Employee Methods
8
9      List<EmployeeVO> GetAllEmployees();
10     EmployeeVO GetEmployee(Guid employeeID);
11     EmployeeVO CreateEmployee(EmployeeVO employee);
12     EmployeeVO UpdateEmployee(EmployeeVO employee);
13     void DeleteEmployee(Guid employeeID);
14
15     #endregion Employee Methods
16
17     #region Training Methods
18
19     List<TrainingVO> GetTrainingForEmployee(Guid employeeID);
20     TrainingVO GetTraining(int trainingID);
21     TrainingVO CreateTraining(TrainingVO training);
22     TrainingVO UpdateTraining(TrainingVO training);
23     void DeleteTraining(int trainingID);
24     void DeleteTrainingForEmployee(Guid employeeID);
25
26     #endregion TrainingMethods
27   }
```

 © 2008 Rick Miller — All Rights Reserved

Referring to Example 20.22 — the IEmployeeTraining interface simply declares the methods required to manage employees and their training.

Example 20.23 gives the code for the EmployeeTrainingRemoteObject class.

20.23 EmployeeTrainingRemoteObject.cs

```
1     using System;
2     using System.Collections.Generic;
3     using EmployeeTraining.VO;
4     using EmployeeTraining.BO;
5
6     public class EmployeeTrainingRemoteObject : MarshalByRefObject, IEmployeeTraining {
7
8       #region Employee Methods
9
10      public List<EmployeeVO> GetAllEmployees(){
11        EmployeeAdminBO bo = new EmployeeAdminBO();
12        return bo.GetAllEmployees();
13      }
14
15      public EmployeeVO GetEmployee(Guid employeeID){
16        EmployeeAdminBO bo = new EmployeeAdminBO();
17        return bo.GetEmployee(employeeID);
18      }
19
20      public EmployeeVO CreateEmployee(EmployeeVO employee){
21        EmployeeAdminBO bo = new EmployeeAdminBO();
22        return bo.CreateEmployee(employee);
23      }
24
25      public EmployeeVO UpdateEmployee(EmployeeVO employee){
26        EmployeeAdminBO bo = new EmployeeAdminBO();
27        return bo.UpdateEmployee(employee);
28      }
29
30      public void DeleteEmployee(Guid employeeID){
31        EmployeeAdminBO bo = new EmployeeAdminBO();
32        bo.DeleteEmployee(employeeID);
33      }
34
35      #endregion Employee Methods
36
37      #region Training Methods
38
39        public TrainingVO CreateTraining(TrainingVO training){
40          EmployeeAdminBO bo = new EmployeeAdminBO();
41          return bo.CreateTraining(training);
42        }
43
44        public TrainingVO GetTraining(int trainingID){
45          EmployeeAdminBO bo = new EmployeeAdminBO();
46          return bo.GetTraining(trainingID);
47        }
48
49        public List<TrainingVO> GetTrainingForEmployee(Guid employeeID){
50          EmployeeAdminBO bo = new EmployeeAdminBO();
51          return bo.GetTrainingForEmployee(employeeID);
52        }
53
54        public TrainingVO UpdateTraining(TrainingVO training){
55          EmployeeAdminBO bo = new EmployeeAdminBO();
56          return bo.UpdateTraining(training);
57        }
58
59        public void DeleteTraining(int trainingID){
60          EmployeeAdminBO bo = new EmployeeAdminBO();
61          bo.DeleteTraining(trainingID);
62        }
63
64        public void DeleteTrainingForEmployee(Guid employeeID){
65          EmployeeAdminBO bo = new EmployeeAdminBO();
66          bo.DeleteTrainingForEmployee(employeeID);
67        }
68
69      #endregion Training Methods
70    }
```

Referring to Example 20.23 — the EmployeeTrainingRemoteObject class extends MarshalByRefObject and implements the methods required by the IEmployeeTraining interface. In this example, the method implementations simply pass the call on to the corresponding EmployeeAdminBO method.

 © 2008 Rick Miller — All Rights Reserved

I did make one change to the EmployeeAdminBO class during this iteration. I modified the DeleteTraining-ForEmployee() method to take a Guid as an argument rather than an EmployeeVO object. This will cut down on network traffic at least somewhat.

Example 20.24 gives the code for the modified EmployeeTrainingServer class.

20.24 EmployeeTrainingServer.cs

```
1    using System;
2    using System.Runtime.Remoting;
3
4    public class EmployeeTrainingServer {
5      public static void Main(){
6        RemotingConfiguration.Configure("EmployeeTrainingServer.exe.config", false);
7        Console.WriteLine("Listening for remote requests. Press any key to exit...");
8        Console.ReadLine();
9      }
10   }
```

Referring to Example 20.24 — this is a whole lot shorter than the last version! This short application simply loads the configuration file. The modified EmployeeTrainingServer.exe.config file is given in Example 20.25.

20.25 EmployeeTrainingServer.exe.config

```
1    <configuration>
2      <configSections>
3        <section name="dataConfiguration"
4                 type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings,
5                       Microsoft.Practices.EnterpriseLibrary.Data, Version=3.1.0.0, Culture=neutral,
6                       PublicKeyToken=b03f5f7f11d50a3a" />
7      </configSections>
8      <dataConfiguration defaultDatabase="Connection String" />
9      <connectionStrings>
10       <add name="Connection String" connectionString="Data Source=(local)\SQLEXPRESS;
11                                     Initial Catalog=EmployeeTraining;Integrated Security=True"
12                                     providerName="System.Data.SqlClient" />
13     </connectionStrings>
14     <system.runtime.remoting>
15       <application>
16         <service>
17          <wellknown mode="Singleton"
18                     type="EmployeeTrainingRemoteObject, EmployeeTrainingRemoteObject"
19                     objectUri="EmployeeTraining" />
20         </service>
21         <channels>
22           <channel ref="tcp" port="8080" />
23         </channels>
24       </application>
25     </system.runtime.remoting>
26   </configuration>
```

Referring to Example 20.25 — the configuration file now sports a `<system.runtime.remoting>` section which gives configuration details about the remote object, its hosting mode (Singleton), and its URI.

Now, to compile the IEmployeeTraining interface, the EmployeeTrainingRemoteObject class, and the EmployeeTrainingServer class, you'll need to make a modification to the MSBuild project file. The modified project file is listed in Example 20.26.

20.26 EmployeeTrainingServer.proj (Mod 2)

```
1    <Project DefaultTargets="CompileApp"
2             xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
3
4       <PropertyGroup>
5         <IncludeDebugInformation>false</IncludeDebugInformation>
6         <BuildDir>build</BuildDir>
7         <LibDir>lib</LibDir>
8         <AppDir>app</AppDir>
9         <RefDir>ref</RefDir>
10        <ConfigDir>config</ConfigDir>
11      </PropertyGroup>
12
13       <ItemGroup>
14         <DAO Include="dao\**\*.cs" />
15         <BO Include="bo\**\*.cs" />
16         <VO Include="vo\**\*.cs" />
17         <APP Include="app\EmployeeTrainingServer.cs" />
18         <REMOTEINTERFACE Include="app\IEmployeeTraining.cs" />
19         <REMOTEOBJECT Include="app\EmployeeTrainingRemoteObject.cs" />
20         <LIB Include="lib\**\*.dll" />
21         <REF Include="ref\**\*.dll" />
22         <CONFIG Include="config\**\*.config" />
23         <EXE Include="app\**\*.exe" />
24       </ItemGroup>
```

 © 2008 Rick Miller — All Rights Reserved

```
25
26      <Target Name="MakeDirs">
27        <MakeDir Directories="$(BuildDir)" />
28        <MakeDir Directories="$(LibDir)" />
29      </Target>
30
31      <Target Name="RemoveDirs">
32        <RemoveDir Directories="$(BuildDir)" />
33        <RemoveDir Directories="$(LibDir)" />
34      </Target>
35
36      <Target Name="Clean"
37              DependsOnTargets="RemoveDirs;MakeDirs">
38      </Target>
39
40       <Target Name="CopyFiles">
41         <Copy
42           SourceFiles="@(CONFIG);@(LIB);@(REF)"
43           DestinationFolder="$(BuildDir)" />
44      </Target>
45
46      <Target Name="CompileVO"
47              Inputs="@(VO)"
48              Outputs="$(LibDir)\VOLib.dll">
49        <Csc Sources="@(VO)"
50            TargetType="library"
51            References="@(REF);@(LIB)"
52            OutputAssembly="$(LibDir)\VOLib.dll">
53        </Csc>
54     </Target>
55
56      <Target Name="CompileDAO"
57              Inputs="@(DAO)"
58              Outputs="$(LibDir)\DAOLib.dll"
59              DependsOnTargets="CompileVO">
60         <Csc Sources="@(DAO)"
61            TargetType="library"
62            References="@(REF);@(LIB)"
63            WarningLevel="0"
64            OutputAssembly="$(LibDir)\DAOLib.dll">
65        </Csc>
66      </Target>
67
68      <Target Name="CompileBO"
69              Inputs="@(BO)"
70              Outputs="$(LibDir)\BOLib.dll"
71              DependsOnTargets="CompileDAO">
72        <Csc Sources="@(BO)"
73            TargetType="library"
74            References="@(REF);@(LIB)"
75            WarningLevel="0"
76            OutputAssembly="$(LibDir)\BOLib.dll">
77        </Csc>
78      </Target>
79
80      <Target Name="CompileApp"
81              Inputs="@(APP);@(REMOTEINTERFACE);@(REMOTEOBJECT)"
82              Outputs="$(BuildDir)\$(MSBuildProjectName).exe;
83                       $(LibDir)\IEmployeeTraining.dll;
84                       $(LibDir)\EmployeeTrainingRemoteObject.dll"
85              DependsOnTargets="CompileBO">
86        <Csc Sources="@(REMOTEINTERFACE)"
87             TargetType="library"
88             References="@(REF);@(LIB)"
89             OutputAssembly="$(LibDir)\IEmployeeTraining.dll">
90         </Csc>
91         <Csc Sources="@(REMOTEOBJECT)"
92             TargetType="library"
93             References="@(REF);@(LIB)"
94             OutputAssembly="$(LibDir)\EmployeeTrainingRemoteObject.dll">
95         </Csc>
96        <Csc Sources="@(APP)"
97             TargetType="exe"
98             References="@(REF);@(LIB)"
99             OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
100       </Csc>
101     </Target>
102
103     <Target Name="Run"
104             DependsOnTargets="CompileApp;CopyFiles">
105       <Exec Command="$(MSBuildProjectName).exe"
```


© 2008 Rick Miller — All Rights Reserved

```
106             WorkingDirectory="$(BuildDir)" />
107      </Target>
108
109  </Project>
```

Referring to Example 20.26 — I've made changes to the <ItemGroup> section and to the <CompileApp> target. To the <ItemGroup> section I added <REMOTEINTERFACE> and <REMOTEOBJECT> items, giving specific names for the corresponding source files. To the <CompileApp> target I added two new <Csc> tasks to compile the IEmployeeTraining and EmployeeTrainingRemoteObject source files.

To compile the EmployeeTrainingServer application, simply execute the CompileApp target by entering the following command-line command:

```
msbuild /t:compileapp
```

If all goes well the EmployeeTrainingServer.exe file will be built and written to the build directory. Change to the build directory and double-click the EmployeeTrainingServer.exe file. You should see an output similar to that shown in Figure 20-44.

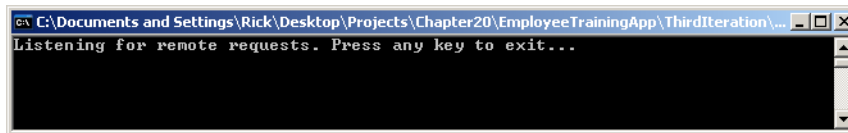


Figure 20-44: EmployeeTrainingServer Running and Ready For Remote Connections

To test the server at this point requires building a suitable remoting client application. I cover this topic in the next section.

The Client Application

In this section I will show you how to build a suitable remoting client application that provides a GUI front-end to the EmployeeTrainingServer application. The GUI-based client application will allow users to manage employees and their training with the help of menus, dialog boxes, and data grid components.

Third Iteration (continued)

The best place to start the client development effort is by setting up the client project folders, building an MSbuild project file, creating a client configuration file, and writing a small client application to test connectivity to the EmployeeTrainingRemoteObject. Table 20-7 lists the design considerations and design decisions for the continuing third iteration.

Check-Off	Design Consideration	Design Decision
	Project directory structure	Create the client application project folders. In the client directory create the app, build, config, and ref subdirectories.
	MSBuild project file	Create an MSBuild project file that will be used to compile and run the client application.
	Client configuration file	Create a configuration file that contains a <system.runtime.remoting> section. The name of the configuration file will be EmployeeTraining-Client.exe.config
	Remoting client application	Start the client application by writing a short program that tests the connection to the remote object.

Table 20-7: Employee Training Client Application — Third Iteration Design Considerations And Decisions (Continued)

Figure 20-45 shows the client project directory structure.

 © 2008 Rick Miller — All Rights Reserved

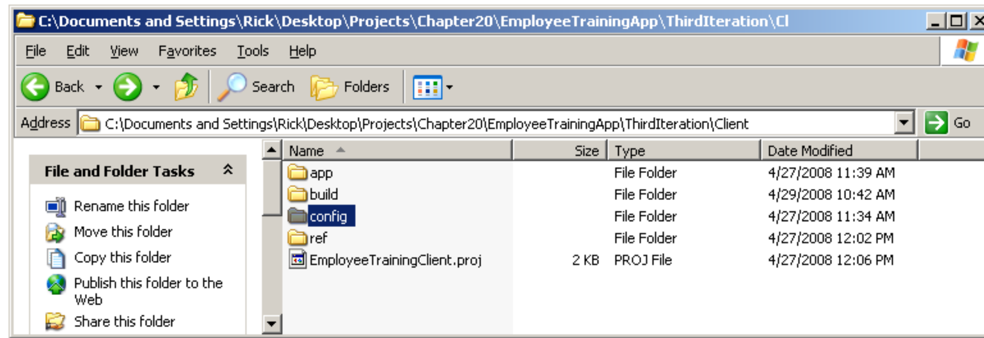


Figure 20-45: Client Project Directory Structure

Referring to Figure 20-45 — the client application source code goes in the app folder. The configuration file resides in the config folder, and the required dlls must be placed in the ref folder. For this iteration you will need the IEmployeeTraining.dll and the VOLib.dll files. You will find these dlls in the server project's lib directory. The client executable file will be built to the build folder and any required dlls will be moved to that location as well.

Example 20.27 gives the code for the EmployeeTrainingClient.proj project file.

20.27 EmployeeTrainingClient.proj

```
1     <Project DefaultTargets="Run"
2              xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
3
4        <PropertyGroup>
5          <IncludeDebugInformation>false</IncludeDebugInformation>
6          <BuildDir>build</BuildDir>
7          <AppDir>app</AppDir>
8          <RefDir>ref</RefDir>
9          <ConfigDir>config</ConfigDir>
10       </PropertyGroup>
11
12        <ItemGroup>
13
14          <APP Include="app\EmployeeTrainingClient.cs" />
15          <REF Include="ref\**\*.dll" />
16          <CONFIG Include="config\**\*.config" />
17          <EXE Include="app\**\*.exe" />
18        </ItemGroup>
19
20        <Target Name="MakeDirs">
21          <MakeDir Directories="$(BuildDir)" />
22        </Target>
23
24        <Target Name="RemoveDirs">
25          <RemoveDir Directories="$(BuildDir)" />
26        </Target>
27
28        <Target Name="Clean"
29                DependsOnTargets="RemoveDirs;MakeDirs">
30        </Target>
31
32         <Target Name="CopyFiles">
33           <Copy
34             SourceFiles="@(CONFIG);@(REF)"
35             DestinationFolder="$(BuildDir)" />
36        </Target>
37
38        <Target Name="CompileApp"
39                Inputs="@(APP)"
40                Outputs="$(BuildDir)\$(MSBuildProjectName).exe"
41                DependsOnTargets="Clean">
42          <Csc Sources="@(APP)"
43               TargetType="exe"
44               References="@(REF)"
45               OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
46          </Csc>
47        </Target>
48
49        <Target Name="Run"
50                DependsOnTargets="CompileApp;CopyFiles">
51          <Exec Command="$(MSBuildProjectName).exe"
```

 © 2008 Rick Miller — All Rights Reserved

```
52              WorkingDirectory="$(BuildDir)" />
53      </Target>
54   </Project>
```

Referring to Example 20.27 — this project file contains <PropertyGroup> and <ItemGroup> sections along with several targets. There are two primary targets: CompileApp and Run. The default project target is specified on line 1 as the Run target. The Run target depends on the CompileApp and CopyFiles targets.

Example 20.28 gives the code for the EmployeeTrainingClient.exe.config configuration file.

20.28 EmployeeTrainingClient.exe.config

```
1   <configuration>
2     <system.runtime.remoting>
3       <application>
4         <client>
5           <wellknown type="IEmployeeTraining, IEmployeeTraining"
6                      url="tcp://localhost:8080/EmployeeTraining" />
7         </client>
8       </application>
9     </system.runtime.remoting>
10  </configuration>
```

Referring to Example 20.28 — the client configuration file has a <system.runtime.remoting> section, which specifies the remote object type and its url.

Example 20.29 gives the code for the EmployeeTrainingClient application.

20.29 EmployeeTrainingClient.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.Remoting;
4   using System.Runtime.Remoting.Channels;
5   using System.Runtime.Remoting.Channels.Tcp;
6   using EmployeeTraining.VO;
7
8   public class EmployeeTrainingClient {
9     public static void Main(){
10     try {
11       RemotingConfiguration.Configure("EmployeeTrainingClient.exe.config", false);
12       WellKnownClientTypeEntry[] client_types = RemotingConfiguration.GetRegisteredWellKnownClientTypes();
13       IEmployeeTraining employee_training =
14            (IEmployeeTraining)Activator.GetObject(typeof(IEmployeeTraining), client_types[0].ObjectUrl );
15       Console.WriteLine("Remote EmployeeTraining object successfully created!");
16       List<EmployeeVO> employee_list = employee_training.GetAllEmployees();
17       foreach(EmployeeVO emp in employee_list){
18         Console.WriteLine(emp.FirstName + " " + emp.MiddleName + " " + emp.LastName);
19       }
20      }catch(Exception e){
21        Console.WriteLine(e);
22      }
23    }
24  }
```

Referring to Example 20.29 — this first short version of the client application tests the connectivity to the remote object. Once it obtains the proxy to the remote object, it calls the GetAllEmployees() method and prints the returned information to the console.

To build and run this application make sure you've copied the required dlls to the client's ref folder and have started the server. Run the msbuild project file's Run target with the following command-line command:

```
msbuild /t:run
```

Also, since the Run target is the default target, you could also simply enter the following command:

```
msbuild
```

Figure 20-46 shows the results of running the first version of the client application.

Fourth Iteration

It's time now in this development iteration to flesh out the final version of the Employee Training client application. As you proceed with development you may find that you'll need to make some changes to the server application in order to accommodate some unforeseen design problems.

A good place to start this development cycle is to sketch out a framework for the client GUI application, implement a piece of it, and continue with testing the server application, as the minimal amount of testing done in the pre-

 © 2008 Rick Miller — All Rights Reserved

```
Projects
  Creating directory "build".
CopyFiles:
  Copying file from "config\EmployeeTrainingClient.exe.config" to "build\EmployeeTrainingClient.exe.config".
  Copying file from "ref\IEmployeeTraining.dll" to "build\IEmployeeTraining.dll".
  Copying file from "ref\VOLib.dll" to "build\VOLib.dll".
Run:
  Remote EmployeeTraining object successfully created!
  Rick Warren Miller
  Patrick J. Condemi
  Steve Jacob Bishop
  Alicia K. Hicks
  Coralie Sarah Powell
  Kyle Victor Miller
  Patrick Tony Condemi
  Dana Lee Condemi
Done Building Project "C:\Documents and Settings\Rick\Desktop\Projects\Chapter20\EmployeeTrainingApp\ThirdIteration\Cli
ent\EmployeeTrainingClient.proj" (default targets).


Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.60
```

Figure 20-46: Running Client Application via the MSBuild Project's Run Target

vious iteration was wholly inadequate. Table 20-8 lists the design considerations and design decisions for the fourth development iteration.

Check-Off	Design Consideration	Design Decision
	Client application	Sketch out a mock-up of the client application GUI and start its implementation. The client application will need to use the EmployeeTraining remote object, so you'll need to pass a reference to the remote object into the client application. This you can do via the client application constructor.
	Application testing	Continue testing the server side components and note any deficiencies.

Table 20-8: Employee Training Client Application — Fourth Iteration Design Considerations And Decisions

Referring to Table 20-8 — these two activities are quite enough to bite off for this iteration. Let's start with a UML diagram of the EmployeeTrainingClient application class, as is shown in Figure 20-47.

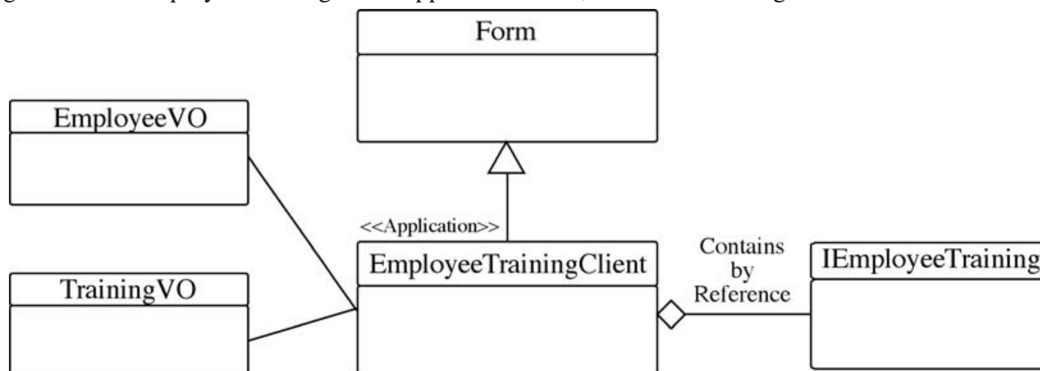


Figure 20-47: EmployeeTrainingClient UML Class Diagram

Referring to Figure 20-47 — the EmployeeTrainingClient class extends the Form class. (System.Windows.Form) It also contains by reference an instance of IEmployeeTraining, and it has a dependency on the EmployeeVO and TrainingVO classes. Thus, if changes are required to the server side components, you'll need to ensure you copy the required dependant dlls into the client project's ref folder before building the client application. The dependent dlls include VOLib.dll and IEmployeeTraining.dll.

Figure 20-48 shows a mock-up sketch of the GUI layout for the EmployeeTrainingClient application.

Referring to Figure 20-48 — the GUI contains a menu with several menu items. Here I've only shown two menu items, but the final application may contain more. DataGridView components are used to display employee and training information. A PictureBox component contains the employee's picture. The components are arranged in a Table-

 © 2008 Rick Miller — All Rights Reserved

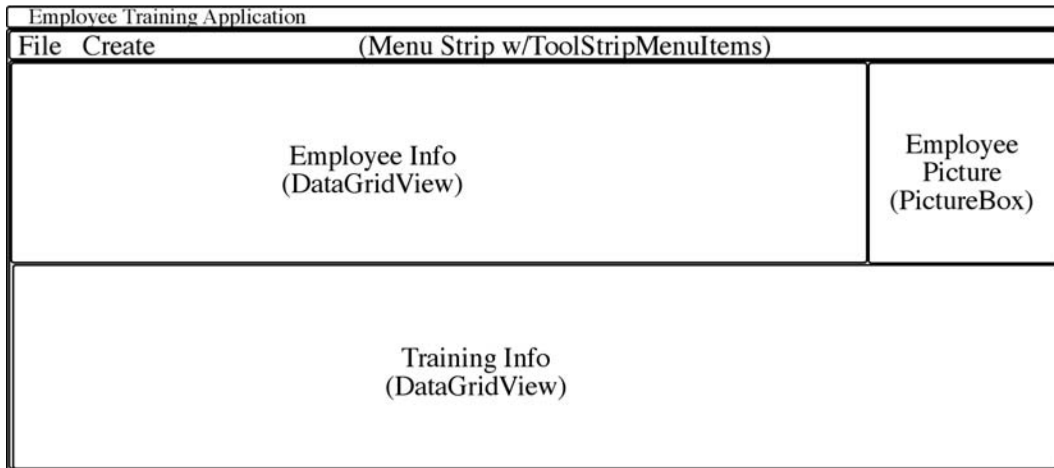


Figure 20-48: Mock-up Sketch of the EmployeeTrainingApplication GUI

LayoutPanel containing two rows and two columns. The employee DataGridView goes into the upper left table layout cell, and the PictureBox is placed in the upper right cell. The training DataGridView is placed in the second row and spans two columns. Example 20.30 gives the code for the partial implementation of this application.

20.30 EmployeeTrainingClient.cs

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4    using System.IO;
5    using System.ComponentModel;
6    using System.Collections.Generic;
7    using System.Runtime.Remoting;
8    using System.Runtime.Remoting.Channels;
9    using System.Runtime.Remoting.Channels.Tcp;
10   using EmployeeTraining.VO;
11
12   public class EmployeeTrainingClient : Form {
13
14     // Constants
15     private const int WINDOW_HEIGHT = 500;
16     private const int WINDOW_WIDTH = 900;
17     private const String WINDOW_TITLE = "Employee Training Application";
18     private const bool DEBUG = true;
19
20     // fields
21     private IEmployeeTraining _employeeTraining = null;
22     private List<EmployeeVO> _employeeList = null;
23     private TableLayoutPanel _tablePanel = null;
24     private DataGridView _employeeGrid = null;
25     private DataGridView _trainingGrid = null;
26     private PictureBox _pictureBox = null;
27
28     public EmployeeTrainingClient(IEmployeeTraining employeeTraining){
29       _employeeTraining = employeeTraining;
30       this.InitializeComponent();
31     }
32
33     private void InitializeComponent(){
34       // setup the menus
35       MenuStrip ms = new MenuStrip();
36
37       ToolStripMenuItem fileMenu = new ToolStripMenuItem("File");
38       ToolStripMenuItem exitMenuItem = new ToolStripMenuItem("Exit", null,
39                                                               new EventHandler(this.ExitProgramHandler));
40
41       ToolStripMenuItem createMenu = new ToolStripMenuItem("Create");
42       ToolStripMenuItem employeeMenuItem = new ToolStripMenuItem("Employee...", null,
43                                                             new EventHandler(this.CreateEmployeeHandler));
44       ToolStripMenuItem trainingMenuItem = new ToolStripMenuItem("Training...", null,
45                                                             new EventHandler(this.CreateTrainingHandler));
46
47       fileMenu.DropDownItems.Add(exitMenuItem);
```

 © 2008 Rick Miller — All Rights Reserved

```
48       ms.Items.Add(fileMenu);
49
50       createMenu.DropDownItems.Add(employeeMenuItem);
51       createMenu.DropDownItems.Add(trainingMenuItem);
52       ms.Items.Add(createMenu);
53
54       // create the table panel
55       _tablePanel = new TableLayoutPanel();
56       _tablePanel.RowCount = 2;
57       _tablePanel.ColumnCount = 2;
58       _tablePanel.Anchor =  AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
59       _tablePanel.Dock = DockStyle.Top;
60       _tablePanel.Height = 400;
61
62       // create and initialize the data grids
63       _employeeGrid = new DataGridView();
64       _employeeGrid.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
65       _employeeGrid.Height = 200;
66       _employeeGrid.Width = 700;
67       _employeeList = _employeeTraining.GetAllEmployees();
68       _employeeGrid.DataSource = _employeeList;
69       _employeeGrid.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
70       _employeeGrid.Click += this.EmployeeGridClickedHandler;
71
72       _trainingGrid = new DataGridView();
73       _trainingGrid.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
74       _trainingGrid.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
75
76       // create picture box
77       _pictureBox = new PictureBox();
78       _pictureBox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
79
80       //add grids to table panel
81       _tablePanel.Controls.Add(_employeeGrid);
82       _tablePanel.Controls.Add(_pictureBox);
83       _tablePanel.Controls.Add(_trainingGrid);
84       _tablePanel.SetColumnSpan(_trainingGrid, 2);
85
86       this.Controls.Add(_tablePanel);
87       ms.Dock = DockStyle.Top;
88       this.MainMenuStrip = ms;
89       this.Controls.Add(ms);
90       this.Height = WINDOW_HEIGHT;
91       this.Width = WINDOW_WIDTH;
92       this.Text = WINDOW_TITLE;
93     }
94
95     /************************************************************
96       Event Handlers
97     *************************************************************/
98     private void ExitProgramHandler(Object sender, EventArgs e){
99       Application.Exit();
100    }
101
102    private void CreateEmployeeHandler(Object sender, EventArgs e){
103      // add code here
104    }
105
106    private void CreateTrainingHandler(Object sender, EventArgs e){
107      // add code here
108    }
109
110    private void EmployeeGridClickedHandler(Object sender, EventArgs e){
111      int selected_row = _employeeGrid.SelectedRows[0].Index;
112      Image employee_picture = _employeeList[selected_row].Picture;
113
114      if(employee_picture !=  null){
115        _pictureBox.Image = employee_picture;
116      }
117      if(DEBUG){ // print some info to the console
118        Console.WriteLine(selected_row);
119        Console.WriteLine(_employeeList[selected_row]);
120      }
121
122      _trainingGrid.DataSource = null;
123      _trainingGrid.DataSource =
124                         _employeeTraining.GetTrainingForEmployee(_employeeList[selected_row].EmployeeID);
125    }
126
127    public static void Main(){
128     try {
```


© 2008 Rick Miller — All Rights Reserved

```
129         RemotingConfiguration.Configure("EmployeeTrainingClient.exe.config", false);
130         WellKnownClientTypeEntry[] client_types = RemotingConfiguration.GetRegisteredWellKnownClientTypes();
131         IEmployeeTraining employee_training =
132               (IEmployeeTraining)Activator.GetObject(typeof(IEmployeeTraining), client_types[0].ObjectUrl );
133         EmployeeTrainingClient client = new EmployeeTrainingClient(employee_training);
134         Application.Run(client);
135        }catch(Exception e){
136          Console.WriteLine(e);
137        }
138      }
139   } // end class definition
```

Referring to Example 20.30 — the EmployeeTrainingClient class extends Form, as expected. It contains two DataGridViews and a PictureBox, which are contained within a TableLayoutPanel in accordance with the mock-up sketch given in Figure 20-48. All menu item event handler methods, with the exception of the File->Exit menu item event handler, are stub methods that will eventually need to be fleshed out.

Let's take a look at the Main() method which begins on line 127. The bulk of the Main() method remains unchanged from the previous iteration. The test code has been removed and replaced with lines 133 and 134. These lines of code create an instance of the EmployeeTrainingClient, passing into the constructor the reference to the remote object, and then calling Application.Run() to kick things off.

Look now at the InitializeComponent() method which begins on line 33. The first thing I do is create and initialize the menu strip and its associated menu items. Next, beginning on line 55, I create and initialize the TableLayoutPanel, followed by the creation and initialization of the DataGridView components. When I create the _employeeGrid, I make a call via the remote object reference _employeeTraining to get a list of all employees. I assign this list to the _employeeList reference and then use this reference to set the _employeeGrid.DataSource property.

So, what happens when the application starts is this: The client application displays a list of employees and their associated data in the employee DataGridView component. When a user clicks on the employee DataGridView, that Click event is handled by the EmployeeGridClickedHandler() method, which begins on line 110. The event handler loads the employee's picture into the PictureBox component as long as the employee's picture is not null. It then loads the employee's training into the training DataGridView by setting its DataSource property via a call to the remote object's GetTrainingForEmployee() method. A click on a DataGridView yields a row index value. This row index value is used to index the _employeeList to retrieve the appropriate EmployeeVO object.

To compile and run this application make sure you've copied the requisite dlls from the server application's lib folder to the client application's ref folder, start the server application, and then from the client application project directory run MSBuild with the default target like so:

```
msbuild
```

If all goes well you'll see the client application window open and it should look something like Figure 20-49.

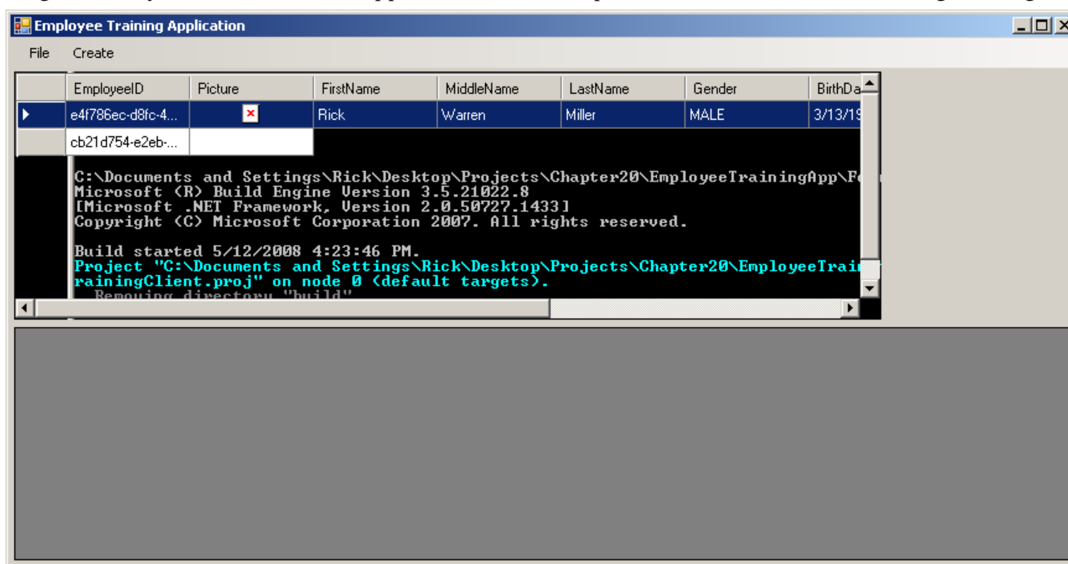


Figure 20-49: EmployeeTrainingClient Initial Display on Startup — Something's Not Quite Right!

 © 2008 Rick Miller — All Rights Reserved

Referring to Figure 20-49 — well, something's amiss! Your display will look different depending on what was behind your application window on startup. Let's try clicking on the first row of the employee information DataGridView and see if the related training will display. Figure 20-50 shows the results. Referring to Figure 20-50 — when I

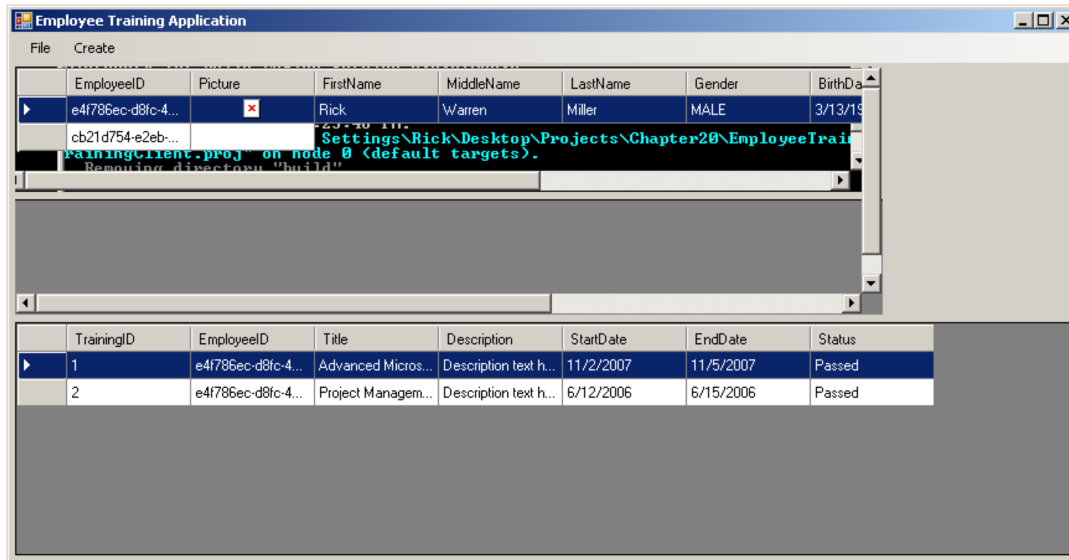


Figure 20-50: Employee's Related Training Shown in Training DataGridView

click the first row (I'm clicking on the gray margin to the left of each row) the related training for that employee shows up in the training DataGridView. However, the first employee has no picture. Let's see what happens when I click on the second row. Figure 20-51 shows the results. (Cross your fingers!)

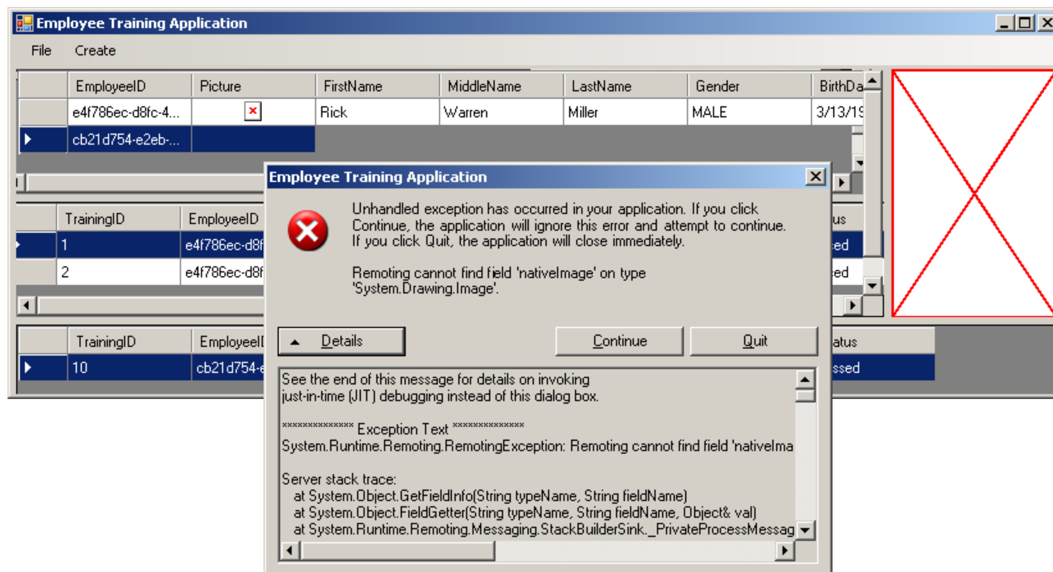


Figure 20-51: Results of Clicking on a Employee with a Picture - a RemotingException is Thrown

Referring to Figure 20-51 — something is obviously wrong! I've received a rather cryptic RemotingException saying: "Remoting cannot find field 'nativeImage' on type System.Drawing.Image." Upon deep investigation around the Internet I finally find the following note buried on the MSDN website for the System.Drawing.Bitmap class:

"The Bitmap class is not accessible across application domains. For example, if you create a

© 2008 Rick Miller — All Rights Reserved

dynamic AppDomain and create several brushes, pens, and bitmaps in that domain, then pass these objects back to the main application domain, you can successfully use the pens and brushes. However, if you call the DrawImage method to draw the marshaled Bitmap, you receive the following exception. Remoting cannot find field "native image" on type "System.Drawing.Image". "

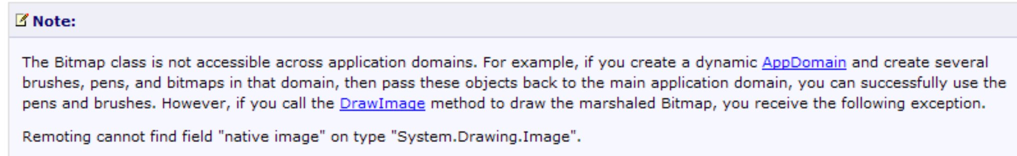


Figure 20-52: Bitmap Class Usage Note

I've also shown the Bitmap note in Figure 20-52. OK, so if you can't transfer an Image across application domains, how are you to transfer the employee's picture? You'll have to do it the old fashioned way — store the employee's picture as an array of bytes. These should transfer across application domains with no problem. To do this will require some changes to the server application. This fix will be the focus of the fifth development iteration.

Fifth Iteration

In the previous development iteration we encountered a problem with transferring the Employee's picture across application domains via .NET remoting. This problem played havoc with the employee DataGridView component. In this iteration I'm going to fix that problem by modifying the server application to hold the employee's picture as an array of bytes. (*i.e.*, a byte[]) To make this fix I'll need to modify two server-side classes: EmployeeVO and EmployeeDAO. I'll also need to modify the EmployeeTrainingClient class to properly handle the modified EmployeeVO class. (You see, it's sweet having an application architecture that lets you zero in on exactly what components need to be modified to implement the fix.)

Table 20-9 gives the design considerations and design decisions for the fifth iteration.

Check-Off	Design Consideration	Design Decision
	Employee picture transfer problem: EmployeeVO class	Modify the EmployeeVO class to hold employee picture data in a byte array.
	Employee picture transfer problem: EmployeeDAO class	Modify the EmployeeDAO class to properly insert the byte array into the tbl_employee.Picture column and to properly populate the EmployeeVO upon retrieval.
	EmployeeTrainingClient class	Modify the EmployeeGridClickedHandler() method to properly handle the modified EmployeeVO class.
	Application testing	Continue with application testing to ensure the changes work. Some of the changes to the DAO will not be tested fully until the next iteration.

Table 20-9: Employee Training Client Application — Fifth Iteration Design Considerations And Decisions

Example 20.31 gives the modified code for the EmployeeVO class.

20.31 EmployeeVO.cs (modified)

```
1     using System;
2
3     namespace EmployeeTraining.VO {
4     [Serializable]
5       public class EmployeeVO : PersonVO {
6
7       // private instance fields
8       private Guid     _employeeID;
9       private byte[] _picturebytes;
```

 © 2008 Rick Miller — All Rights Reserved

```
10
11     //default constructor
12     public EmployeeVO(){}
13
14     public EmployeeVO(Guid employeeid, String firstName, String middleName, String lastName,
15                   Sex gender, DateTime birthday):base(firstName, middleName, lastName, gender, birthday){
16        EmployeeID = employeeid;
17     }
18
19     // public properties
20     public Guid EmployeeID {
21       get { return _employeeID;  }
22       set { _employeeID = value; }
23     }
24
25     public byte[] Picture {
26       get { return _picturebytes; }
27       set { _picturebytes = value; }
28     }
29
30     public override String ToString(){
31       return (EmployeeID + " " + base.ToString());
32     }
33   } // end EmployeeVO class
34   } // end namespace
```

Referring to Example 20.31 — I've made three changes to this class. First, I removed the `using` System.Drawing directive since I no longer need to use the System.Drawing.Image class. Second, I removed the _picture field and replaced it with the _picturebytes field which is of type byte array (byte[]). Lastly, I changed the Picture property to reflect it's new type and to get and set the _picturebytes field.

Example 20.32 gives the code for the modified EmployeeDAO class.

20.32 EmployeeDAO.cs (modified)

```
1    using System;
2    using System.IO;
3    using System.Data;
4    using System.Data.Common;
5    using System.Data.Sql;
6    using System.Data.SqlTypes;
7    using System.Data.SqlClient;
8    using System.Collections.Generic;
9    //using System.Drawing;
10   //using System.Drawing.Imaging;
11   using EmployeeTraining.VO;
12
13   using Microsoft.Practices.EnterpriseLibrary.Common;
14   using Microsoft.Practices.EnterpriseLibrary.Data;
15   using Microsoft.Practices.EnterpriseLibrary.Data.Sql;
16
17   namespace EmployeeTraining.DAO {
18     public class EmployeeDAO : BaseDAO {
19
20      private bool debug = true;
21
22       //List of column identifiers used in perpared statements
23       private const String EMPLOYEE_ID = "@employee_id";
24       private const String FIRST_NAME = "@first_name";
25       private const String MIDDLE_NAME = "@middle_name";
26       private const String LAST_NAME = "@last_name";
27       private const String BIRTHDAY = "@birthday";
28       private const String GENDER = "@gender";
29       private const String PICTURE = "@picture";
30
31       private const String SELECT_ALL_COLUMNS =
32         "SELECT employeeid, firstname, middlename, lastname, birthday, gender, picture ";
33
34       private const String SELECT_ALL_EMPLOYEES =
35         SELECT_ALL_COLUMNS +
36         "FROM tbl_employee ";
37
38       private const String SELECT_EMPLOYEE_BY_EMPLOYEE_ID =
39         SELECT_ALL_EMPLOYEES +
40         "WHERE employeeid = " + EMPLOYEE_ID;
41
42
43       private const String INSERT_EMPLOYEE =
44         "INSERT INTO tbl_employee " +
45             "(EmployeeID, FirstName, MiddleName, LastName, Birthday, Gender, Picture) " +
46          "VALUES (" + EMPLOYEE_ID + ", " + FIRST_NAME + ", " + MIDDLE_NAME + ", " + LAST_NAME + ", " +
47                       BIRTHDAY + ", " + GENDER + ", " + PICTURE + ")";
```


© 2008 Rick Miller — All Rights Reserved

```
48
49        private const String UPDATE_EMPLOYEE =
50          "UPDATE tbl_employee " +
51          "SET FirstName = " + FIRST_NAME + ", MiddleName = " + MIDDLE_NAME + ", LastName = " + LAST_NAME +
52                 ", Birthday = " + BIRTHDAY + ", Gender = " + GENDER + ", Picture = " + PICTURE + " " +
53          "WHERE EmployeeID = " + EMPLOYEE_ID;
54
55        private const String DELETE_EMPLOYEE =
56          "DELETE FROM tbl_employee " +
57          "WHERE EmployeeID = " + EMPLOYEE_ID;
58
59        /************************************
60                  Returns a List<EmployeeVO> object
61              *************************************/
62        public List<EmployeeVO> GetAllEmployees(){
63          DbCommand command = DataBase.GetSqlStringCommand(SELECT_ALL_EMPLOYEES);
64          return this.GetEmployeeList(command);
65        }
66
67        /***********************************************
68                 Returns an EmployeeVO object given a valid employeeid
69              ************************************************/
70        public EmployeeVO GetEmployee(Guid employeeid){
71          DbCommand command = null;
72          try{
73            command = DataBase.GetSqlStringCommand(SELECT_EMPLOYEE_BY_EMPLOYEE_ID);
74            DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
75          }catch(Exception e){
76            Console.WriteLine(e);
77          }
78          return this.GetEmployee(command);
79        }
80
81        /******************************************************
82                Inserts an employee given a fully-populated EmployeeVO object
83             *******************************************************/
84        public EmployeeVO InsertEmployee(EmployeeVO employee){
85          try{
86            employee.EmployeeID = Guid.NewGuid();
87            DbCommand command = DataBase.GetSqlStringCommand(INSERT_EMPLOYEE);
88            DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employee.EmployeeID);
89            DataBase.AddInParameter(command, FIRST_NAME, DbType.String, employee.FirstName);
90            DataBase.AddInParameter(command, MIDDLE_NAME, DbType.String, employee.MiddleName);
91            DataBase.AddInParameter(command, LAST_NAME, DbType.String, employee.LastName);
92            DataBase.AddInParameter(command, BIRTHDAY, DbType.DateTime, employee.BirthDay);
93            switch(employee.Gender){
94              case EmployeeVO.Sex.MALE: DataBase.AddInParameter(command, GENDER, DbType.String, "M");
95                   break;
96              case EmployeeVO.Sex.FEMALE: DataBase.AddInParameter(command, GENDER, DbType.String, "F");
97                   break;
98            }
99
100           if(employee.Picture != null){
101            if(debug){ Console.WriteLine("Inserting picture!"); }
102             if(debug){
103               for(int i=0; i<employee.Picture.Length; i++){
104                 Console.Write(employee.Picture[i]);
105               }
106             } // end if debug
107            DataBase.AddInParameter(command, PICTURE, DbType.Binary, employee.Picture);
108            if(debug){ Console.WriteLine("Picture inserted, I think!"); }
109           }
110           DataBase.ExecuteNonQuery(command);
111         }catch(Exception e){
112           Console.WriteLine(e);
113         }
114         return this.GetEmployee(employee.EmployeeID);
115       }
116
117       /*******************************************************
118                Updates a row in the tbl_employee table given the fully-populated
119                EmployeeVO object.
120             *******************************************************/
121       public EmployeeVO UpdateEmployee(EmployeeVO employee){
122         try {
123           DbCommand command = DataBase.GetSqlStringCommand(UPDATE_EMPLOYEE);
124           DataBase.AddInParameter(command, FIRST_NAME, DbType.String, employee.FirstName);
125           DataBase.AddInParameter(command, MIDDLE_NAME, DbType.String, employee.MiddleName);
126           DataBase.AddInParameter(command, LAST_NAME, DbType.String, employee.LastName);
127           DataBase.AddInParameter(command, BIRTHDAY, DbType.DateTime, employee.BirthDay);
128           switch(employee.Gender){
```

 © 2008 Rick Miller — All Rights Reserved

```
129          case EmployeeVO.Sex.MALE: DataBase.AddInParameter(command, GENDER, DbType.String, "M");
130               break;
131          case EmployeeVO.Sex.FEMALE: DataBase.AddInParameter(command, GENDER, DbType.String, "F");
132               break;
133        }
134        if(employee.Picture != null){
135         if(debug){ Console.WriteLine("Inserting picture!"); }
136          if(debug){
137            for(int i=0; i<employee.Picture.Length; i++){
138              Console.Write(employee.Picture[i]);
139            }
140          } // end if debug
141         DataBase.AddInParameter(command, PICTURE, DbType.Binary, employee.Picture);
142         if(debug){ Console.WriteLine("Picture inserted, I think!"); }
143        }
144        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employee.EmployeeID);
145        DataBase.ExecuteNonQuery(command);
146    }catch(Exception e){
147      Console.WriteLine(e);
148    }
149    return this.GetEmployee(employee.EmployeeID);
150    }
151
152   /*********************************************************
153           Deletes a row from the tbl_employee table given an employee id.
154       *******************************************************/
155   public void DeleteEmployee(Guid employeeid){
156     try{
157       DbCommand command = DataBase.GetSqlStringCommand(DELETE_EMPLOYEE);
158       DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
159       DataBase.ExecuteNonQuery(command);
160     }catch(Exception e){
161       Console.WriteLine(e);
162     }
163   }
164
165    /*************************************************
166              Private utility method that executes the given DbCommand
167              and returns a fully-populated EmployeeVO object
168         ************************************************/
169    private EmployeeVO GetEmployee(DbCommand command){
170      EmployeeVO empVO = null;
171      IDataReader reader = null;
172      try {
173        reader = DataBase.ExecuteReader(command);
174        if(reader.Read()){
175          empVO = this.FillInEmployeeVO(reader);
176        }
177      }catch(Exception e){
178        Console.WriteLine(e);
179      }finally {
180        base.CloseReader(reader);
181      }
182      return empVO;
183    }
184
185    /*****************************************************
186             GetEmployeeList() - returns a List<EmployeeVO> object
187         ***************************************************/
188    private List<EmployeeVO> GetEmployeeList(DbCommand command){
189      IDataReader reader = null;
190      List<EmployeeVO> employee_list = new List<EmployeeVO>();
191      try{
192        reader = DataBase.ExecuteReader(command);
193        while(reader.Read()){
194          EmployeeVO empVO = this.FillInEmployeeVO(reader);
195          employee_list.Add(empVO);
196        }
197      }catch(Exception e){
198        Console.WriteLine(e);
199      }finally{
200        base.CloseReader(reader);
201      }
202      return employee_list;
203    }
204
205    /*********************************************************
206             Private utility method that populates an EmployeeVO object from
207             data read from the IDataReader object
208        *********************************************************/
209    private EmployeeVO FillInEmployeeVO(IDataReader reader){
```

 © 2008 Rick Miller — All Rights Reserved

```
210        EmployeeVO empVO = new EmployeeVO();
211        empVO.EmployeeID = reader.GetGuid(0);
212        empVO.FirstName = reader.GetString(1);
213        empVO.MiddleName = reader.GetString(2);
214        empVO.LastName = reader.GetString(3);
215        empVO.BirthDay = reader.GetDateTime(4);
216        String gender = reader.GetString(5);
217        switch(gender){
218          case "M" : empVO.Gender = EmployeeVO.Sex.MALE;
219                     break;
220          case "F" : empVO.Gender = EmployeeVO.Sex.FEMALE;
221                     break;
222        }
223        if(!reader.IsDBNull(6)){
224          int buffersize = 5000;
225          int startindex = 0;
226          Byte[] byte_array = new Byte[buffersize];
227          MemoryStream ms = new MemoryStream();
228          long retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
229          while(retval > 0){
230            ms.Write(byte_array, 0, byte_array.Length);
231            startindex += buffersize;
232            retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
233           }
234          empVO.Picture = ms.ToArray();
235        }
236        return empVO;
237      }
238
239    } // end EmployeeDAO definition
240  } // end namespace
```

Referring to Example 20.32 — I removed the `using` System.Drawing and `using` System.Drawing.Imaging directives, and made modifications to the InsertEmployee(), UpdateEmployee(), and FillInEmployeeVO() methods to properly handle the insertion and retrieval of a byte_array. Actually, a byte array was already being inserted and retrieved from the database. The only changes I made involved the elimination of the image conversion step. The code is actually simplified now that there's no need to convert an image into an array of bytes. However, this conversion will now need to be performed in the client application when an employee picture is selected for insertion.

Example 20.33 gives the code for the modified EmployeeGridClickedHandler() method which is found in the EmployeeTrainingClient class.

20.33 EmployeeGridClickedHandler() Method (modified)

```
1    private void EmployeeGridClickedHandler(Object sender, EventArgs e){
2        int selected_row = _employeeGrid.SelectedRows[0].Index;
3        byte[] pictureBytes = _employeeList[selected_row].Picture;
4
5        if(pictureBytes !=  null){
6          MemoryStream ms = new MemoryStream();
7          ms.Write(pictureBytes, 0, pictureBytes.Length);
8          _pictureBox.Image = new Bitmap(ms);
9        } else {
10          _pictureBox.Image = null;
11         }
12       Console.WriteLine(selected_row);
13       Console.WriteLine(_employeeList[selected_row]);
14
15       _trainingGrid.DataSource = null;
16       _trainingGrid.DataSource =
17                      _employeeTraining.GetTrainingForEmployee(_employeeList[selected_row].EmployeeID);
18
19     }
```

Referring to Example 20.33 — the selected employee's Picture array is assigned to the pictureBytes reference. The `if` statement beginning on line 5 checks to see if the pictureBytes reference is not null. If it's not null, the pictureBytes array is written to a MemoryStream object, which is then used to create a Bitmap object.

Let's test these changes before proceeding further. You'll need to recompile the server application and copy the IEmployeeTraining.dll and VOLib.dll files to the client's ref folder. Start the server and then run the client. Figure 20-53 shows the client application with an employee's picture displayed in the PictureBox.

Referring to Figure 20-53 — it seems the byte array is the way to go. You can also see a portion of each employee's picture (those that have one) in the corresponding cell under the Picture column. However, I'm not sure I want the employee picture in the DataGridView as it would make each row too high. I'll fix this in the next development iteration as well as add the ability to create and edit employees and their associated training.

 © 2008 Rick Miller — All Rights Reserved

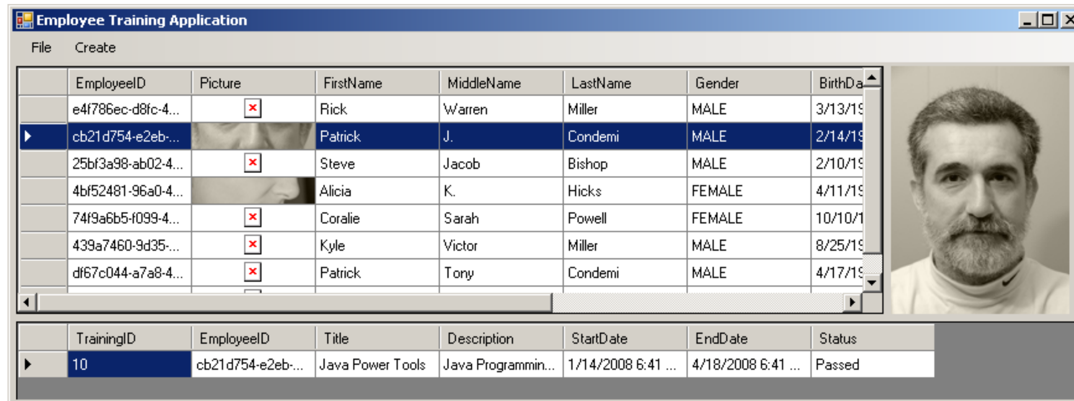


Figure 20-53: EmployeeTrainingClient Application with Employee's Picture Displayed in the PictureBox

Sixth Iteration

Now that an employee's data, including their image data, can be successfully transferred across the network, it's time to add more features to the EmployeeTrainingClient application. One thing I'll do will be to customize the DataGridViews and hide a few of the columns I don't want to display. I'll also add the ability to create, edit, and delete employees and training records. I'll use separate forms to enter and edit employee and training data. Table 20-10 lists the design considerations and design decisions for the sixth iteration.

Check-Off	Design Consideration	Design Decision
	Hide unwanted DataGridView columns.	The columns displayed in a DataGridView correspond to public properties of the EmployeeVO and TrainingVO classes. For the employee's DataGridView I'll hide the EmployeeID, Picture, FullName, and FullNameAndAge columns. For the training DataGridView I'll hide the EmployeeID and TrainingID columns.

	Application menus.	I think I'll do a redesign here and rename the Create menu and call it the Edit menu instead. To the Edit menu I'll add the following menu items:
		`Create Employee...`
		`Edit Employee`
		`---------`
		`Create Training...`
		`Edit Training...`
		`---------`
		`Delete Employee...`
		`Delete Training...`
		I'll need to do some menu manipulation while the application is running so I will move the declaration of the menu items out of the InitializeComponent() method so that I have access to them throughout the application. I'll also need to use a MessageBox to give users the chance to change their mind about deleting an employee or a training record.

	Employee form	I'll need to create a data entry form suitable for use both to create a new employee and to edit an existing employee. (Note: I could create and edit via the DataGridView but I'll leave that as an exercise for you!)
	Training form	I'll also need a data entry form suitable for use both to create and edit training records.

Table 20-10: Employee Training Client Application — Sixth Iteration Design Considerations And Decisions

 © 2008 Rick Miller — All Rights Reserved

I think I'll start by designing and implementing the data entry forms. Figure 20-54 shows the mock-up for the employee data entry form.

Employee Form
First Name:
Middle Name:
Last Name:
Birthday:
Gender:
Male
Female
Clear
Load Picture
Submit

Figure 20-54: Employee Form Mock-up

Referring to Figure 20-54 — the employee form will contain the components required to enter and edit employee information. The components I'll need to use include Labels, TextBoxes, RadioButtons and a GroupBox, Buttons, and a PictureBox. I'll arrange the components with the help of several TableLayoutPanels and a FlowLayoutPanel.

I'll need a way to set and get the values of each data entry component. I'll make this possible by adding read-write properties to the employee form. Example 20.34 gives the code for the EmployeeForm class.

20.34 EmployeeForm.cs

```
1    using System;
2    using System.Drawing;
3    using System.Windows.Forms;
4    using EmployeeTraining.VO;
5
6    public class EmployeeForm : Form {
7      // constants
8      private const int WINDOW_HEIGHT = 300;
9      private const int WINDOW_WIDTH = 550;
10
11     // fields
12     private TableLayoutPanel _mainTablePanel;
13     private TableLayoutPanel _infoTablePanel;
14     private FlowLayoutPanel _buttonPanel;
15     private PictureBox _pictureBox;
16     private Label _firstNameLabel;
17     private Label _middleNameLabel;
18     private Label _lastNameLabel;
19     private Label _birthdayLabel;
20     private Label _genderLabel;
21     private TextBox _firstNameTextBox;
22     private TextBox _middleNameTextBox;
23     private TextBox _lastNameTextBox;
24     private DateTimePicker _birthdayPicker;
25     private GroupBox _genderBox;
26     private RadioButton _maleRadioButton;
27     private RadioButton _femaleRadioButton;
28     private Button _clearButton;
29     private Button _loadPictureButton;
30     private Button _submitButton;
31     private OpenFileDialog _dialog;
32     private bool _createMode;
33
34
35     // public properties -
36     public String FirstName {
37       get { return _firstNameTextBox.Text; }
38       set { _firstNameTextBox.Text = value; }
39     }
40
41     public String MiddleName {
42       get { return _middleNameTextBox.Text; }
43       set { _middleNameTextBox.Text = value; }
44     }
45
46     public String LastName {
47       get { return _lastNameTextBox.Text; }
48       set { _lastNameTextBox.Text = value; }
49     }
```

 © 2008 Rick Miller — All Rights Reserved

```
50
51     public DateTime Birthday {
52       get { return _birthdayPicker.Value; }
53       set { _birthdayPicker.Value = value; }
54     }
55
56     public Image Picture {
57       get { return _pictureBox.Image; }
58       set { _pictureBox.Image = value; }
59     }
60
61     public PersonVO.Sex Gender {
62       get { return this.RadioButtonToSexEnum(); }
63       set { this.SetRadioButton(value); }
64     }
65
66     public bool CreateMode {
67       get { return _createMode; }
68       set { _createMode = value; }
69     }
70
71     public bool SubmitOK {
72       set { _submitButton.Enabled = value; }
73     }
74
75     public EmployeeForm(EmployeeTrainingClient externalHandler){
76       this.InitializeComponent(externalHandler);
77     }
78
79     private void InitializeComponent(EmployeeTrainingClient externalHandler){
80      _mainTablePanel = new TableLayoutPanel();
81      _mainTablePanel.RowCount = 2;
82      _mainTablePanel.ColumnCount = 2;
83      _mainTablePanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right
84                             | AnchorStyles.Left;
85      _mainTablePanel.Height = 500;
86      _mainTablePanel.Width = 700;
87      _infoTablePanel = new TableLayoutPanel();
88      _infoTablePanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right
89                             | AnchorStyles.Left;
90      _infoTablePanel.RowCount = 2;
91      _infoTablePanel.ColumnCount = 2;
92      _infoTablePanel.Height = 200;
93      _infoTablePanel.Width = 400;
94      _buttonPanel = new FlowLayoutPanel();
95      _buttonPanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right | AnchorStyles.Left;
96      _buttonPanel.Width = 500;
97      _buttonPanel.Height = 200;
98
99      _pictureBox = new PictureBox();
100     _pictureBox.Height = 200;
101     _pictureBox.Width = 200;
102     _pictureBox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right | AnchorStyles.Left;
103
104     _firstNameLabel = new Label();
105     _firstNameLabel.Text = "First Name:";
106     _middleNameLabel = new Label();
107     _middleNameLabel.Text = "Middle Name:";
108     _lastNameLabel = new Label();
109     _lastNameLabel.Text = "Last Name:";
110     _birthdayLabel = new Label();
111     _birthdayLabel.Text = "Birthday";
112     _genderLabel = new Label();
113     _genderLabel.Text = "Gender";
114     _firstNameTextBox = new TextBox();
115     _firstNameTextBox.Width = 200;
116     _middleNameTextBox = new TextBox();
117     _middleNameTextBox.Width = 200;
118     _lastNameTextBox = new TextBox();
119     _lastNameTextBox.Width = 200;
120     _birthdayPicker = new DateTimePicker();
121     _genderBox = new GroupBox();
122     _genderBox.Text = "Gender";
123     _genderBox.Height = 75;
124     _genderBox.Width = 200;
125     _maleRadioButton = new RadioButton();
126     _maleRadioButton.Text = "Male";
127     _maleRadioButton.Checked = true;
128     _maleRadioButton.Location = new Point(10, 20);
129     _femaleRadioButton = new RadioButton();
130     _femaleRadioButton.Text = "Female";
```

 © 2008 Rick Miller — All Rights Reserved

```
131     _femaleRadioButton.Location = new Point(10, 40);
132     _genderBox.Controls.Add(_maleRadioButton);
133     _genderBox.Controls.Add(_femaleRadioButton);
134     _clearButton = new Button();
135     _clearButton.Text = "Clear";
136     _clearButton.Click += this.ClearButtonHandler;
137     _loadPictureButton = new Button();
138     _loadPictureButton.Text = "Load Picture";
139     _loadPictureButton.AutoSize = true;
140     _loadPictureButton.Click += this.LoadPictureButtonHandler;
141     _submitButton = new Button();
142     _submitButton.Text = "Submit";
143     _submitButton.Click += externalHandler.EmployeeSubmitButtonHandler;
144     _submitButton.Enabled = false;
145
146     _infoTablePanel.SuspendLayout();
147     _infoTablePanel.Controls.Add(_firstNameLabel);
148     _infoTablePanel.Controls.Add(_firstNameTextBox);
149     _infoTablePanel.Controls.Add(_middleNameLabel);
150     _infoTablePanel.Controls.Add(_middleNameTextBox);
151     _infoTablePanel.Controls.Add(_lastNameLabel);
152     _infoTablePanel.Controls.Add(_lastNameTextBox);
153     _infoTablePanel.Controls.Add(_birthdayLabel);
154     _infoTablePanel.Controls.Add(_birthdayPicker);
155     _infoTablePanel.Controls.Add(_genderLabel);
156     _infoTablePanel.Controls.Add(_genderBox);
157     _infoTablePanel.Dock = DockStyle.Top;
158
159     _buttonPanel.SuspendLayout();
160     _buttonPanel.Controls.Add(_clearButton);
161     _buttonPanel.Controls.Add(_loadPictureButton);
162     _buttonPanel.Controls.Add(_submitButton);
163
164     _mainTablePanel.SuspendLayout();
165     _mainTablePanel.Controls.Add(_pictureBox);
166     _mainTablePanel.Controls.Add(_infoTablePanel);
167     _mainTablePanel.Controls.Add(_buttonPanel);
168     _mainTablePanel.SetColumnSpan(_buttonPanel, 2);
169
170      this.SuspendLayout();
171      this.Controls.Add(_mainTablePanel);
172      this.Width = WINDOW_WIDTH;
173      this.Height = WINDOW_HEIGHT;
174      this.Text = "Employee Form";
175      _infoTablePanel.ResumeLayout();
176      _buttonPanel.ResumeLayout();
177      _mainTablePanel.ResumeLayout();
178      this.ResumeLayout();
179      _dialog = new OpenFileDialog();
180     _dialog.FileOk += this.LoadPicture;
181    }
182
183    private void ClearButtonHandler(Object sender, EventArgs e){
184      this.ClearFields();
185      _submitButton.Enabled = false;
186    }
187
188    private void LoadPictureButtonHandler(Object sender, EventArgs e){
189      _dialog.ShowDialog();
190    }
191
192    private void LoadPicture(Object sender, EventArgs e){
193      String filename = _dialog.FileName;
194      _pictureBox.Image = new Bitmap(filename);
195      _submitButton.Enabled = true;
196    }
197
198    public void ClearFields(){
199      _firstNameTextBox.Text = String.Empty;
200      _middleNameTextBox.Text = String.Empty;
201      _lastNameTextBox.Text = String.Empty;
202      _maleRadioButton.Checked = true;
203      _birthdayPicker.Value = DateTime.Now;
204      _pictureBox.Image = null;
205    }
206
207    private PersonVO.Sex RadioButtonToSexEnum(){
208      PersonVO.Sex gender = PersonVO.Sex.MALE;
209      if(_maleRadioButton.Checked){
210        gender = PersonVO.Sex.MALE;
211      }else{
```

 © 2008 Rick Miller — All Rights Reserved

```
212          gender = PersonVO.Sex.FEMALE;
213        }
214      return gender;
215    }
216
217    private void SetRadioButton(PersonVO.Sex gender){
218      if(gender == PersonVO.Sex.MALE){
219        _maleRadioButton.Checked = true;
220      }else{
221        _femaleRadioButton.Checked = true;
222      }
223    }
224
225  } // end class definition
```

Referring to Example 20.34 — most of the code is straightforward. The class contains several constants, fields, properties, and event handlers. The _submitButton.Click event is handled by the EmployeeTrainingClient.EmployeeSubmitButtonHandler() method. The _clearButton and _loadPictureButton Click events are handled by local event handlers.

Note that most of the properties consist of simple get and set statements, however, the Gender property's get and set call methods to perform the heavy lifting. The reason for this is that the radio button settings must be translated into Person.Sex enumeration values and vice versa. The CreateMode property is used to indicate whether the form is used to create a new employee or edit an existing employee.

Figure 20-55 shows the mock-up for the training form.

Figure 20-55: Training Form Mock-up

Referring to Figure 20-55 — the training form is built similar to the employee form. It will contain the data entry components required to create and edit an employee training record. It too uses TableLayoutPanels and a FlowLayoutPanel to arrange the components. Example 20.35 shows the code for the TrainingForm class.

20.35 TrainingForm .cs

```
1    using System;
2    using System.Drawing;
3    using System.Windows.Forms;
4    using System.Collections.Generic;
5    using EmployeeTraining.VO;
6
7    public class TrainingForm : Form {
8       // constants
9       private const int WINDOW_HEIGHT = 300;
10      private const int WINDOW_WIDTH = 450;
11      private const bool DEBUG = true;
12
13      // fields
14      private TableLayoutPanel _mainTablePanel;
15      private TableLayoutPanel _infoTablePanel;
16      private FlowLayoutPanel _buttonPanel;
17      private Label _titleLabel;
18      private Label _descriptionLabel;
19      private Label _startDateLabel;
20      private Label _endDateLabel;
21      private Label _statusLabel;
22      private TextBox _titleTextBox;
23      private TextBox _descriptionTextBox;
24      private DateTimePicker _startDatePicker;
25      private DateTimePicker _endDatePicker;
26      private GroupBox _statusGroupBox;
27      private RadioButton _passedRadioButton;
28      private RadioButton _failedRadioButton;
```


© 2008 Rick Miller — All Rights Reserved

```
29      private Button _clearButton;
30      private Button _submitButton;
31      private bool _createMode;
32
33      // public properties -
34      public String Title {
35        get { return _titleTextBox.Text; }
36        set { _titleTextBox.Text = value; }
37      }
38
39      public String Description {
40        get { return _descriptionTextBox.Text; }
41        set { _descriptionTextBox.Text = value; }
42      }
43
44      public DateTime StartDate {
45        get { return _startDatePicker.Value; }
46        set { _startDatePicker.Value = value; }
47      }
48
49      public DateTime EndDate {
50        get { return _endDatePicker.Value; }
51        set { _endDatePicker.Value = value; }
52      }
53
54      public TrainingVO.TrainingStatus Status {
55        get { return this.RadioButtonToTrainingStatusEnum(); }
56        set { this.SetRadioButton(value); }
57      }
58
59      public bool CreateMode {
60        get { return _createMode; }
61        set { _createMode = value; }
62      }
63
64      public TrainingForm(EmployeeTrainingClient externalHandler){
65        this.InitializeComponent(externalHandler);
66      }
67
68      private void InitializeComponent(EmployeeTrainingClient externalHandler){
69        _mainTablePanel = new TableLayoutPanel();
70        _mainTablePanel.RowCount = 2;
71        _mainTablePanel.ColumnCount = 1;
72        _mainTablePanel.Height = 400;
73        _mainTablePanel.Width = 500;
74        _mainTablePanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left
75                               | AnchorStyles.Right;
76
77        _infoTablePanel = new TableLayoutPanel();
78        _infoTablePanel.RowCount = 5;
79        _infoTablePanel.ColumnCount = 2;
80        _infoTablePanel.Height = 200;
81        _infoTablePanel.Width = 300;
82        _infoTablePanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left
83                                 | AnchorStyles.Right;
84
85        _buttonPanel = new FlowLayoutPanel();
86        _buttonPanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
87
88        _titleLabel = new Label();
89        _titleLabel.Text = "Title:";
90        _descriptionLabel = new Label();
91        _descriptionLabel.Text = "Description:";
92        _startDateLabel = new Label();
93        _startDateLabel.Text = "Start Date:";
94        _endDateLabel = new Label();
95        _endDateLabel.Text = "End Date";
96        _statusLabel = new Label();
97        _statusLabel.Text = "Status";
98        _titleTextBox = new TextBox();
99        _titleTextBox.Width = 300;
100       _descriptionTextBox = new TextBox();
101       _descriptionTextBox.Width = 300;
102       _startDatePicker = new DateTimePicker();
103       _endDatePicker = new DateTimePicker();
104       _statusGroupBox = new GroupBox();
105       _statusGroupBox.Height = 75;
106       _statusGroupBox.Width = 300;
107       _passedRadioButton = new RadioButton();
108       _passedRadioButton.Text = "Passed";
109       _passedRadioButton.Checked = true;
```

 © 2008 Rick Miller — All Rights Reserved

```
110       _passedRadioButton.Location = new Point(10, 10);
111       _failedRadioButton = new RadioButton();
112       _failedRadioButton.Text = "Failed";
113       _failedRadioButton.Location = new Point(10, 30);
114       _clearButton = new Button();
115       _clearButton.Text = "Clear";
116       _clearButton.Click += this.ClearButtonHandler;
117       _submitButton = new Button();
118       _submitButton.Text = "Submit";
119       _submitButton.Click += externalHandler.TrainingSubmitButtonHandler;
120
121       _statusGroupBox.Controls.Add(_passedRadioButton);
122       _statusGroupBox.Controls.Add(_failedRadioButton);
123
124       _infoTablePanel.SuspendLayout();
125       _infoTablePanel.Controls.Add(_titleLabel);
126       _infoTablePanel.Controls.Add(_titleTextBox);
127       _infoTablePanel.Controls.Add(_descriptionLabel);
128       _infoTablePanel.Controls.Add(_descriptionTextBox);
129       _infoTablePanel.Controls.Add(_startDateLabel);
130       _infoTablePanel.Controls.Add(_startDatePicker);
131       _infoTablePanel.Controls.Add(_endDateLabel);
132       _infoTablePanel.Controls.Add(_endDatePicker);
133       _infoTablePanel.Controls.Add(_statusLabel);
134       _infoTablePanel.Controls.Add(_statusGroupBox);
135
136       _buttonPanel.Controls.Add(_clearButton);
137       _buttonPanel.Controls.Add(_submitButton);
138
139
140       _mainTablePanel.SuspendLayout();
141       _mainTablePanel.Controls.Add(_infoTablePanel);
142       _mainTablePanel.Controls.Add(_buttonPanel);
143
144       this.SuspendLayout();
145       this.Controls.Add(_mainTablePanel);
146       this.Height = WINDOW_HEIGHT;
147       this.Width = WINDOW_WIDTH;
148       this.Text = "Training Form";
149       _infoTablePanel.ResumeLayout();
150       _mainTablePanel.ResumeLayout();
151       this.ResumeLayout();
152     }
153
154     private TrainingVO.TrainingStatus RadioButtonToTrainingStatusEnum(){
155      TrainingVO.TrainingStatus status = TrainingVO.TrainingStatus.Passed;
156      if(_passedRadioButton.Checked){
157        status = TrainingVO.TrainingStatus.Passed;
158      }else{
159          status = TrainingVO.TrainingStatus.Failed;
160      }
161      return status;
162    }
163
164     private void ClearButtonHandler(Object sender, EventArgs e){
165       this.ClearFields();
166     }
167
168     public void ClearFields(){
169       _titleTextBox.Text = String.Empty;
170       _descriptionTextBox.Text = String.Empty;
171       _startDatePicker.Value = DateTime.Now;
172       _endDatePicker.Value = DateTime.Now;
173       _passedRadioButton.Checked = true;
174     }
175
176     private void SetRadioButton(TrainingVO.TrainingStatus status){
177      if(status == TrainingVO.TrainingStatus.Passed){
178        _passedRadioButton.Checked = true;
179      }else{
180        _failedRadioButton.Checked = true;
181      }
182    }
183  } // end class definition
```

Example 20.36 gives the code for the revised EmployeeTrainingClient class.

20.36 EmployeeTrainingClient.cs (revised)

```
1    using System;
2    using System.Windows.Forms;
3    using System.Drawing;
4    using System.Drawing.Imaging;
```


© 2008 Rick Miller — All Rights Reserved

```
5    using System.IO;
6    using System.ComponentModel;
7    using System.Collections.Generic;
8    using System.Runtime.Remoting;
9    using System.Runtime.Remoting.Channels;
10   using System.Runtime.Remoting.Channels.Tcp;
11   using EmployeeTraining.VO;
12
13   public class EmployeeTrainingClient : Form {
14
15     // Constants
16     private const int WINDOW_HEIGHT = 500;
17     private const int WINDOW_WIDTH = 900;
18     private const String WINDOW_TITLE = "Employee Training Application";
19     private const bool DEBUG = true;
20
21     // fields
22     private MenuStrip _ms;
23     private ToolStripMenuItem _fileMenu;
24     private ToolStripMenuItem _exitMenuItem;
25     private ToolStripMenuItem _editMenu;
26     private ToolStripMenuItem _createEmployeeMenuItem;
27     private ToolStripMenuItem _createTrainingMenuItem;
28     private ToolStripMenuItem _editEmployeeMenuItem;
29     private ToolStripMenuItem _editTrainingMenuItem;
30     private ToolStripMenuItem _deleteEmployeeMenuItem;
31     private ToolStripMenuItem _deleteTrainingMenuItem;
32     private IEmployeeTraining _employeeTraining = null;
33     private List<EmployeeVO> _employeeList = null;
34     private List<TrainingVO> _trainingList = null;
35     private TableLayoutPanel _tablePanel = null;
36     private DataGridView _employeeGrid = null;
37     private DataGridView _trainingGrid = null;
38     private PictureBox _pictureBox = null;
39     private EmployeeForm _employeeForm;
40     private TrainingForm _trainingForm;
41
42     public EmployeeTrainingClient(IEmployeeTraining employeeTraining){
43       _employeeTraining = employeeTraining;
44       this.InitializeComponent();
45     }
46
47     private void InitializeComponent(){
48       // setup the menus
49       _ms = new MenuStrip();
50
51       _fileMenu = new ToolStripMenuItem("File");
52       _exitMenuItem = new ToolStripMenuItem("Exit", null, new EventHandler(this.ExitProgramHandler));
53
54       _editMenu = new ToolStripMenuItem("Edit");
55       _createEmployeeMenuItem = new ToolStripMenuItem("Create Employee...", null,
56                                    new EventHandler(this.CreateEmployeeHandler));
57       _createTrainingMenuItem = new ToolStripMenuItem("Create Training...", null,
58                                    new EventHandler(this.CreateTrainingHandler));
59       _editEmployeeMenuItem = new ToolStripMenuItem("Edit Employee...", null,
60                                    new EventHandler(this.EditEmployeeHandler));
61       _editEmployeeMenuItem.Enabled = false;
62       _editTrainingMenuItem = new ToolStripMenuItem("Edit Training...", null,
63                               new EventHandler(this.EditTrainingHandler));
64       _editTrainingMenuItem.Enabled = false;
65       _deleteEmployeeMenuItem = new ToolStripMenuItem("Delete Employee...", null,
66                                  new EventHandler(this.DeleteEmployeeHandler));
67       _deleteEmployeeMenuItem.Enabled = false;
68       _deleteTrainingMenuItem = new ToolStripMenuItem("Delete Training...", null,
69                                  new EventHandler(this.DeleteTrainingHandler));
70       _deleteTrainingMenuItem.Enabled = false;
71
72       _fileMenu.DropDownItems.Add(_exitMenuItem);
73       _ms.Items.Add(_fileMenu);
74
75       _editMenu.DropDownItems.Add(_createEmployeeMenuItem);
76       _editMenu.DropDownItems.Add(_createTrainingMenuItem);
77       _editMenu.DropDownItems.Add("-");
78       _editMenu.DropDownItems.Add(_editEmployeeMenuItem);
79       _editMenu.DropDownItems.Add(_editTrainingMenuItem);
80       _editMenu.DropDownItems.Add("-");
81       _editMenu.DropDownItems.Add(_deleteEmployeeMenuItem);
82       _editMenu.DropDownItems.Add(_deleteTrainingMenuItem);
83       _ms.Items.Add(_editMenu);
84
85       // create the table panel
```

 © 2008 Rick Miller — All Rights Reserved

```
86      _tablePanel = new TableLayoutPanel();
87      _tablePanel.RowCount = 2;
88      _tablePanel.ColumnCount = 2;
89      _tablePanel.Anchor =  AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
90      _tablePanel.Dock = DockStyle.Top;
91      _tablePanel.Height = 400;
92
93      // create and initialize the data grids
94      _employeeGrid = new DataGridView();
95      _employeeGrid.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
96      _employeeGrid.Height = 200;
97      _employeeGrid.Width = 700;
98      _employeeList = _employeeTraining.GetAllEmployees();
99      _employeeGrid.DataSource = _employeeList;
100     _employeeGrid.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
101     _employeeGrid.Click += this.EmployeeGridClickedHandler;
102     _employeeGrid.DataBindingComplete += this.EmployeeGridDataBindingCompleteHandler;
103
104     _trainingGrid = new DataGridView();
105     _trainingGrid.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
106     _trainingGrid.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
107     _trainingGrid.DataBindingComplete += this.TrainingGridDataBindingCompleteHandler;
108
109
110     _trainingList = _employeeTraining.GetTrainingForEmployee(_employeeList[0].EmployeeID);
111     _trainingGrid.DataSource = _trainingList;
112
113     // create picture box
114     _pictureBox = new PictureBox();
115     _pictureBox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
116
117
118     //add grids to table panel
119     _tablePanel.Controls.Add(_employeeGrid);
120     _tablePanel.Controls.Add(_pictureBox);
121     _tablePanel.Controls.Add(_trainingGrid);
122     _tablePanel.SetColumnSpan(_trainingGrid, 2);
123
124     this.Controls.Add(_tablePanel);
125     _ms.Dock = DockStyle.Top;
126     this.MainMenuStrip = _ms;
127     this.Controls.Add(_ms);
128     this.Height = WINDOW_HEIGHT;
129     this.Width = WINDOW_WIDTH;
130     this.Text = WINDOW_TITLE;
131     _employeeForm = new EmployeeForm(this);
132     _employeeForm.Visible = false;
133     _trainingForm = new TrainingForm(this);
134     _trainingForm.Visible = false;
135   }
136
137   /************************************************************
138     Event Handlers
139   ************************************************************/
140   private void ExitProgramHandler(Object sender, EventArgs e){
141     Application.Exit();
142   }
143
144   private void CreateEmployeeHandler(Object sender, EventArgs e){
145     _employeeForm.CreateMode = true;
146     _employeeForm.SubmitOK = false;
147     _employeeForm.ClearFields();
148     _employeeForm.ShowDialog();
149   }
150
151   private void CreateTrainingHandler(Object sender, EventArgs e){
152     _trainingForm.CreateMode = true;
153     _trainingForm.ClearFields();
154     _trainingForm.ShowDialog();
155   }
156
157   private void EditEmployeeHandler(Object sender, EventArgs e){
158     _employeeForm.ClearFields();
159     _employeeForm.SubmitOK = true;
160     _employeeForm.CreateMode = false;
161     EmployeeVO vo = _employeeList[_employeeGrid.SelectedRows[0].Index];
162     _employeeForm.FirstName = vo.FirstName;
163     _employeeForm.MiddleName = vo.MiddleName;
164     _employeeForm.LastName = vo.LastName;
165     _employeeForm.Birthday = vo.BirthDay;
166     _employeeForm.Gender = vo.Gender;
```

 © 2008 Rick Miller — All Rights Reserved

```
167       MemoryStream ms = new MemoryStream();
168       if(vo.Picture != null) {
169         ms.Write(vo.Picture, 0, vo.Picture.Length);
170         _employeeForm.Picture = new Bitmap(ms);
171       }
172       _employeeForm.ShowDialog();
173     }
174 
175     private void EditTrainingHandler(Object sender, EventArgs e){
176       _trainingForm.CreateMode = false;
177       TrainingVO vo = _trainingList[_trainingGrid.SelectedRows[0].Index];
178       _trainingForm.Title = vo.Title;
179       _trainingForm.Description = vo.Description;
180       _trainingForm.StartDate = vo.StartDate;
181       _trainingForm.EndDate = vo.EndDate;
182       _trainingForm.Status = vo.Status;
183       _trainingForm.ShowDialog();
184     }
185 
186     private void EmployeeGridClickedHandler(Object sender, EventArgs e){
187       int selected_row = _employeeGrid.SelectedRows[0].Index;
188       byte[] pictureBytes = _employeeList[selected_row].Picture;
189 
190       if(pictureBytes !=  null){
191         MemoryStream ms = new MemoryStream();
192         ms.Write(pictureBytes, 0, pictureBytes.Length);
193         _pictureBox.Image = new Bitmap(ms);
194       } else {
195           _pictureBox.Image = null;
196         }
197       Console.WriteLine(selected_row);
198       Console.WriteLine(_employeeList[selected_row]);
199 
200       _trainingGrid.DataSource = null;
201       _trainingList = _employeeTraining.GetTrainingForEmployee(_employeeList[selected_row].EmployeeID);
202       _trainingGrid.DataSource = _trainingList;
203       if(_trainingList.Count > 0){
204         _trainingGrid.Rows[0].Selected = true;
205         _editTrainingMenuItem.Enabled = true;
206         _deleteTrainingMenuItem.Enabled = true;
207       } else {
208         _editTrainingMenuItem.Enabled = false;
209         _deleteTrainingMenuItem.Enabled = false;
210       }
211 
212       if(DEBUG){
213         foreach(EmployeeVO emp in _employeeList){
214           Console.WriteLine(emp.FirstName + " " + emp.LastName);
215         }
216       }
217     }
218 
219     private void EmployeeGridDataBindingCompleteHandler(Object sender, EventArgs e){
220       _employeeGrid.Columns["Picture"].Visible = false;
221       _employeeGrid.Columns["FullName"].Visible = false;
222       _employeeGrid.Columns["FullNameAndAge"].Visible = false;
223       _employeeGrid.Columns["Age"].ReadOnly = true;
224       _employeeGrid.Columns["Age"].ToolTipText = "Read Only!";
225       _employeeGrid.Columns["EmployeeID"].Visible = false;
226       if(_employeeList.Count > 0){
227         _employeeGrid.Rows[0].Selected = true;
228         this.EmployeeGridClickedHandler(this, new EventArgs());
229         _editEmployeeMenuItem.Enabled = true;
230         _deleteEmployeeMenuItem.Enabled = true;
231       }
232     }
233 
234     private void TrainingGridDataBindingCompleteHandler(Object sender, EventArgs e){
235       _trainingGrid.Columns["TrainingID"].Visible = false;
236       _trainingGrid.Columns["EmployeeID"].Visible = false;
237       if(_trainingList.Count > 0){
238         _trainingGrid.Rows[0].Selected = true;
239         _editTrainingMenuItem.Enabled = true;
240         _deleteTrainingMenuItem.Enabled = true;
241       }
242     }
243 
244     public void EmployeeSubmitButtonHandler(Object sender, EventArgs e){
245       if(_employeeForm.CreateMode){ // creating new employee
246         EmployeeVO vo = new EmployeeVO();
247         vo.FirstName = _employeeForm.FirstName;
```

 © 2008 Rick Miller — All Rights Reserved

```
248        vo.MiddleName = _employeeForm.MiddleName;
249        vo.LastName = _employeeForm.LastName;
250        vo.BirthDay = _employeeForm.Birthday;
251        MemoryStream ms = new MemoryStream();
252        _employeeForm.Picture.Save(ms, ImageFormat.Tiff);
253        vo.Picture = ms.ToArray();
254        vo.Gender = _employeeForm.Gender;
255        _employeeTraining.CreateEmployee(vo);
256        _employeeForm.Visible = false;
257        _employeeList = _employeeTraining.GetAllEmployees();
258        _employeeGrid.DataSource = _employeeList;
259        _employeeForm.ClearFields();
260     }else{ // editing new employee
261         EmployeeVO vo = _employeeList[_employeeGrid.SelectedRows[0].Index];
262         vo.FirstName = _employeeForm.FirstName;
263         vo.MiddleName = _employeeForm.MiddleName;
264         vo.LastName = _employeeForm.LastName;
265         vo.BirthDay = _employeeForm.Birthday;
266         MemoryStream ms = new MemoryStream();
267         _employeeForm.Picture.Save(ms, ImageFormat.Tiff);
268         vo.Picture = ms.ToArray();
269         vo.Gender = _employeeForm.Gender;
270         _employeeTraining.UpdateEmployee(vo);
271         _employeeForm.Visible = false;
272         _employeeList = _employeeTraining.GetAllEmployees();
273         _employeeGrid.DataSource = _employeeList;
274         _employeeForm.ClearFields();
275
276     }
277    }
278
279    public void TrainingSubmitButtonHandler(Object sender, EventArgs e){
280      if(_trainingForm.CreateMode){
281        TrainingVO vo = new TrainingVO();
282        int selected_row = _employeeGrid.SelectedRows[0].Index;
283        vo.EmployeeID = _employeeList[selected_row].EmployeeID;
284        vo.Title = _trainingForm.Title;
285        vo.Description = _trainingForm.Description;
286        vo.StartDate = _trainingForm.StartDate;
287        vo.EndDate = _trainingForm.EndDate;
288        vo.Status = _trainingForm.Status;
289        _employeeTraining.CreateTraining(vo);
290        _trainingGrid.DataSource = null;
291        _trainingGrid.DataSource = _employeeTraining.GetTrainingForEmployee(vo.EmployeeID);
292        _trainingForm.Visible = false;
293        _trainingForm.ClearFields();
294      }else {
295        TrainingVO vo = _trainingList[_trainingGrid.Rows[0].Index];
296        vo.Title = _trainingForm.Title;
297        vo.Description = _trainingForm.Description;
298        vo.StartDate = _trainingForm.StartDate;
299        vo.EndDate = _trainingForm.EndDate;
300        vo.Status = _trainingForm.Status;
301        _employeeTraining.UpdateTraining(vo);
302        _trainingGrid.DataSource = null;
303        _trainingGrid.DataSource = _employeeTraining.GetTrainingForEmployee(vo.EmployeeID);
304        _trainingForm.Visible = false;
305        _trainingForm.ClearFields();
306      }
307    }
308
309    private void DeleteEmployeeHandler(Object sender, EventArgs e){
310      DialogResult result = MessageBox.Show("Are you sure? Click OK to delete, “ +
311                                            “or Cancel to return to the application.",
312                                             "Warning!", MessageBoxButtons.OKCancel, MessageBoxIcon.Warning);
313      if(result == DialogResult.OK){
314        int selected_row = _employeeGrid.SelectedRows[0].Index;
315        _employeeTraining.DeleteEmployee(_employeeList[selected_row].EmployeeID);
316        _employeeGrid.DataSource = null;
317        _employeeList = _employeeTraining.GetAllEmployees();
318        _employeeGrid.DataSource = _employeeList;
319        if(_employeeList.Count > 0){
320          _employeeGrid.Rows[0].Selected = true;
321          this.EmployeeGridClickedHandler(this, new EventArgs());
322          _editEmployeeMenuItem.Enabled = true;
323          _deleteEmployeeMenuItem.Enabled = true;
324        }
325      }
326    }
327
328    private void DeleteTrainingHandler(Object sender, EventArgs e){
```

 © 2008 Rick Miller — All Rights Reserved

```
329       DialogResult result = MessageBox.Show("Are you sure? Click OK to delete, " +
330                                              "or Cancel to return to the application.",
331                                             "Warning!", MessageBoxButtons.OKCancel, MessageBoxIcon.Warning);
332       if(result == DialogResult.OK){
333         int selected_row = _trainingGrid.SelectedRows[0].Index;
334         _employeeTraining.DeleteTraining(_trainingList[selected_row].TrainingID);
335         _trainingGrid.DataSource = null;
336         int selected_employee = _employeeGrid.SelectedRows[0].Index;
337         _trainingList =
338                    _employeeTraining.GetTrainingForEmployee(_employeeList[selected_employee].EmployeeID);
339         _trainingGrid.DataSource = _trainingList;
340         if(_trainingList.Count > 0){
341           _trainingGrid.Rows[0].Selected = true;
342           _editTrainingMenuItem.Enabled = true;
343          _deleteTrainingMenuItem.Enabled = true;
344         }
345       }
346    }
347
348    public static void Main(){
349     try {
350       RemotingConfiguration.Configure("EmployeeTrainingClient.exe.config", false);
351       WellKnownClientTypeEntry[] client_types = RemotingConfiguration.GetRegisteredWellKnownClientTypes();
352       IEmployeeTraining employee_training =
353            (IEmployeeTraining)Activator.GetObject(typeof(IEmployeeTraining), client_types[0].ObjectUrl );
354       EmployeeTrainingClient client = new EmployeeTrainingClient(employee_training);
355       Application.Run(client);
356      }catch(Exception e){
357        Console.WriteLine(e);
358      }
359    }
360  } // end class definition
```

Referring to Example 20.36 — well, there's a lot going on here but it should be easy to follow the code. First, I've moved the declarations for the menu and its menu items into the fields area so I can have access to menu items when I need to manipulate them. "What will I be doing?" you ask. Well, for one thing, I want to disable the "Edit Employee..." and "Delete Employee..." menu choices when there are no employees to edit or delete. I also want to do the same for the "Edit Training..." and "Delete Training..." menu choices.

I would like to focus your attention on a few areas of the code worth special mention. First, before I can hide any columns, I must wait until the DataGridView controls have been properly data bound. Data binding takes place when I assign a data source to a DataGridView's DataSource property. When data binding is complete the control fires the DataBindingComplete event. The column-hiding code for the _employeeGrid is placed in the EmployeeGridDataBindingCompleteHandler() method, which begins on line 219. To get an employee's picture to load into the _pictureBox and their associated training records to display in the _trainingGrid, I make an explicit call to the EmployeeGridClickedHandler() method on line 228.

The _trainingGrid.DataBindingComplete event is handled by the TrainingGridDataBindingCompleteHandler() method which begins on line 234. I place the column-hiding code for the _trainingGrid in this method.

Compiling And Running The Modified EmployeeTrainingClient Project

I placed the EmployeeForm.cs and TrainingForm.cs files in the project's app directory. To compile these files along with the EmployeeTrainingClient.cs file, I need to make a minor change to the EmployeeTrainingClient.proj file. Example 20.37 gives the modified project file.

20.37 EmployeeTrainingClient.proj (modified)

```
1    <Project DefaultTargets="Run"
2            xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
3
4       <PropertyGroup>
5         <IncludeDebugInformation>false</IncludeDebugInformation>
6         <BuildDir>build</BuildDir>
7         <AppDir>app</AppDir>
8         <RefDir>ref</RefDir>
9         <ConfigDir>config</ConfigDir>
10      </PropertyGroup>
11
12       <ItemGroup>
13
14         <APP Include="app\**\*.cs" />
15         <REF Include="ref\**\*.dll" />
16         <CONFIG Include="config\**\*.config" />
17         <EXE Include="app\**\*.exe" />
```


© 2008 Rick Miller — All Rights Reserved

```
18        </ItemGroup>
19
20        <Target Name="MakeDirs">
21          <MakeDir Directories="$(BuildDir)" />
22        </Target>
23
24        <Target Name="RemoveDirs">
25          <RemoveDir Directories="$(BuildDir)" />
26        </Target>
27
28        <Target Name="Clean"
29                DependsOnTargets="RemoveDirs;MakeDirs">
30        </Target>
31
32         <Target Name="CopyFiles">
33           <Copy
34             SourceFiles="@(CONFIG);@(REF)"
35             DestinationFolder="$(BuildDir)" />
36        </Target>
37
38        <Target Name="CompileApp"
39                Inputs="@(APP)"
40                Outputs="$(BuildDir)\$(MSBuildProjectName).exe"
41                DependsOnTargets="Clean">
42          <Csc Sources="@(APP)"
43               TargetType="exe"
44               References="@(REF)"
45               OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
46          </Csc>
47        </Target>
48
49        <Target Name="Run"
50                DependsOnTargets="CompileApp;CopyFiles">
51          <Exec Command="$(MSBuildProjectName).exe"
52                WorkingDirectory="$(BuildDir)" />
53        </Target>
54    </Project>
```

Referring to Example 20.37 — the change appears on line 14 where I've specified the <APP> item to include all the source files found in the project's app folder.

To compile and run the EmployeeTrainingClient project, make sure the server is up and running, change to the EmployeeTrainingClient project directory, and enter the following command-line command:

```
msbuild
```

This executes the default build target. If all goes well you'll see the application window appear. Figure 20-56 shows the main application window with the Edit menu extended to show the new menu items.

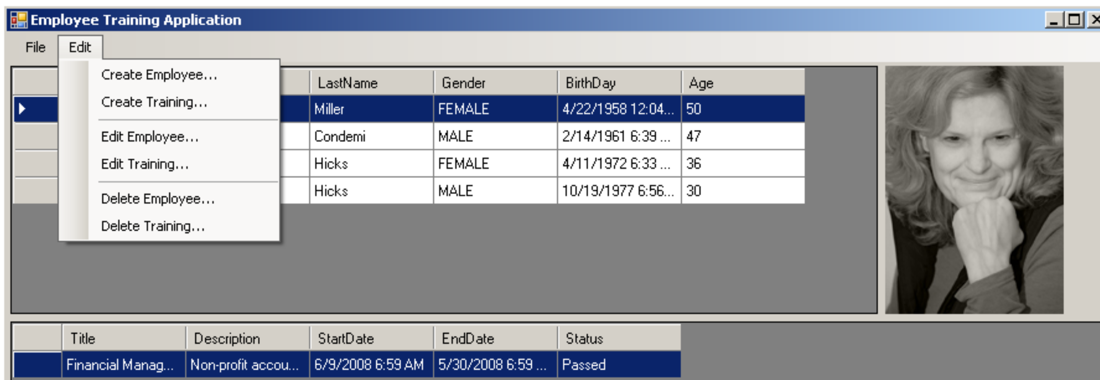


Figure 20-56: Main Application Window with Edit Menu Open to Reveal Revised Menu Structure

Figure 20-57 shows how the Edit menu looks when some of the menu items are disabled.

Referring to Figure 20-57 — Bill Hicks has no training so the "Edit Training..." and "Delete Training..." menu items are disabled.

To create a new employee select Edit->Create Employee... to open the employee form, as is shown in Figure 20-58. Referring to Figure 20-58 — the employee form is cleared when creating a new employee and its Submit button is disabled. To enable the Submit button you need to load a picture. Figure 20-59 shows how the employee form looks fully populated. Figure 20-60 shows how the training form looks empty and fully populated.

 © 2008 Rick Miller — All Rights Reserved

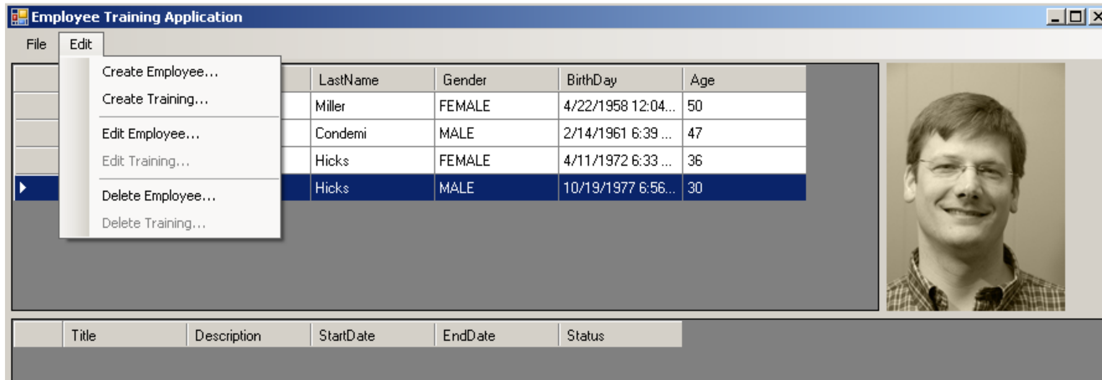


Figure 20-57: Edit Menu Items Disabled

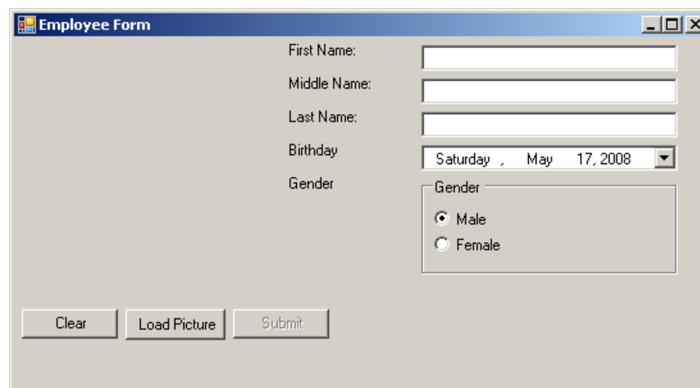


Figure 20-58: Empty Employee Data Entry Form

Figure 20-59: Employee Form Fully Populate and Submit Button Enabled

Where To Go From Here

The DataGridView control is extremely powerful and in the EmployeeTrainingClient application I don't come close to tapping its full potential. So, for starters, I recommend you explore its capabilities further by spending some time on MSDN and researching its members. For example, it's not necessary to have separate data entry forms to enter and edit employee and training data. You can create new DataGridView rows programmatically and edits made to data contained therein are reflected in the bound data source. I've put some code in the EmployeeTrainingClient application that shows how the EmployeeVO objects contained in the _employeeList are changed automatically when you edit an _employeeGrid column. (See Example 20.36 lines 212 - 216)

© 2008 Rick Miller — All Rights Reserved

Figure 20-60: Training Form Empty and Filled

Regarding the database side of things, although I covered a lot of ground, I omitted topics such as normal forms and mapping tables. I'll leave you to explore these and other database topics on your own. Having seen the employee training project developed from start to finish should have filled your head with so many ideas that they are falling out of your ears!

Summary

Relational databases hold data in *tables*. Table *columns* are specified to be of a particular *data type*. Table data is contained in *rows*. Structured Query Language (SQL) is used to *create*, *manipulate*, and *delete* relational database objects and data. SQL contains three sub-languages: Data Definition Language (DDL) which is used to create databases, tables, views, and other database objects; Data Manipulation Language (DML) which is used to create, manipulate, and delete the data contained within a database; and Data Control Language (DCL) which is used to grant or revoke user rights and privileges on database objects.

Different database makers are free to extend SQL to suit their needs so there's no guarantee of SQL portability between different databases.

One or more table columns can be designated as a *primary key* whose value is unique for each row inserted into that table. Related tables can be created by including the primary key of one table as a *foreign key* in the related table.

The select command can be used to construct complex queries involving multiple related tables. One table is joined to another to form a temporary table. There are many different types of *join* operations, but the most common one is an inner join, which is the default join condition provided by Microsoft SQL Server.

Inner joins are made possible through the use of foreign keys. A *foreign key* is a column in a table that contains a value that is used as a primary key in another table. A table can be related to many other tables by including multiple foreign keys. Specify a foreign key by adding a foreign key constraint to a particular table using the alter command.

Use *database scripts* to ease database development. Scripts that create the database, tables, constraints, and test data let you work at the speed of light.

Approach the design and implementation of complex database applications in an iterative fashion. Structure the design of your application in such a way as to make changing the application as painless as possible. A tiered approach to application design allows you to quickly identify and correct problems or make application modifications when you realize your design needs to be changed.

Transfer complex data types as byte arrays (byte[]) and convert them into the appropriate type at the other end.

Skill-Building Exercises

1. **Programming Drill:** Compile and execute the sixth iteration version of the employee training application presented in this chapter.

© 2008 Rick Miller — All Rights Reserved

2. **UML Documentation Drill:** Create a UML sequence diagram that traces the execution of method calls starting from the EmployeeTrainingServerRemoteObject.

3. **API Research:** Visit Microsoft's Patterns and Practices developer center and research the Enterprise Library Data Access Application Block. (DAAB). [http://www.codeplex.com/entlib]

4. **Relational Database Design:** Procure a good book on relational database design theory and practice and read it from front to back.

5. **Normal Forms:** Relational database designers use the concept of *normal forms* to help guide their design decisions. Study the topic of normal forms paying particular attention to the differences between 1st, 2nd, and 3rd normal forms. Also, do some research on when it's sometimes a good idea to denormalize a database.

6. **Relational Relationships:** When associating relational database tables you can have 1-to-N (a.k.a. 1-to-Many), N-to-1 (a.k.a. Many-to-1), and N-to-N (a.k.a. Many-to-Many). Research each of these table relationship types so you have an understanding of when and why each should be used.

Suggested Projects

1. **Program Modification:** Modify the Employee Training application so that it can store and display an employee's address and contact information. Enable the application to associate one or more employee's with one or more addresses. (Hint: N-to-N) Enable the application to associate each employee with one or more contact numbers, email addresses, etc. (Hint: 1-to-N)

2. **Program Modification:** Currently, in the Employee Training application, due to the 1-to-N relationship between the tbl_employee and tbl_employee_training table, if more than one employee takes the same class there's a lot of repetitive data stored in the database. This can lead to a loss of data integrity, especially if a user enters the title or description training data differently for different employees who attended the same training. Modify the application so that training class data is entered into the database only once. Enable users to select from a list of available training when entering an employee's training information. (Hint: This will require an N-to-N relationship between an employee and a training record. In between there will be a linking table which contains the start date and end date, status, and any other occurrence-specific data.)

3. **Sex Offender Database:** Design and build an application that let's you register and track sex offenders. Some ideas for data you might want to maintain include name, addresses, aliases, behavior practices, and employment.

4. **Non-Profit Fund Raising Tracker Application:** Design and build an application that lets you record and track the contributions made to a non-profit organization. The database should record donor information including address and contact information, and amounts donated and when.

Self-Test Questions

1. In what type of structure do relational databases store data.

2. What's the purpose of a primary key?

3. What's the purpose of a foreign key?

4. What are the names of SQL's three sublanguages? What's the purpose of each sublanguage?

© 2008 Rick Miller — All Rights Reserved

5. (T/F) All database makers implement SQL the exact same way.

6. What's the purpose of the Data Access Object (DAO) application layer?

7. What's the purpose of the Business Object (BO) application layer?

8. What's the purpose of a Value Object (VO).

9. Into what data structure should you convert complex data types before serializing them and transferring then over the network?

10. How should you approach the design and implementation of a complex, multitiered application?

References

Microsoft Patterns and Practices Developer Center. [http://www.codeplex.com/entlib]

Microsoft Developer Network (MSDN) *.NET Framework 3.0 and3.5 Reference Documentation* [www.msdn.com]

Candace C. Fleming & Barbara von Halle. Handbook of Relational Database Design, Addison-Wesley Professional, 1989, ISBN: 0-201-11434-8

Notes

© 2008 Rick Miller — All Rights Reserved

 © 2008 Rick Miller — All Rights Reserved

Part V: Advanced Concepts

© 2008 Rick Miller — All Rights Reserved

© 2008 Rick Miller — All Rights Reserved

Chapter 21

Contax T / Kodak Tri-X

Picnic At The Washington Canoe Club

Operator Overloading

Learning Objectives

- *State the purpose of operator overloading*
- *List the overloadable operators*
- *State why it is important to preserve expected operator semantics*
- *List the operators that must be overloaded in pairs*
- *State the requirements for overloading unary operators*
- *Implement the true and false operators*
- *State the requirements for overloading binary operators*
- *Implement explicit type conversion operators*
- *State which operators you get for free when you overload the binary arithmetic operators*
- *Demonstrate your ability to overload C# operators*

© 2008 Rick Miller — All Rights Reserved

Introduction

C# allows you to add meaning to certain language operators so they behave in expected ways when applied to user-defined types. Adding meaning to operators in this fashion is referred to as *operator overloading*.

You have already seen many examples of operator overloading in action. Consider for a moment the equality operator ==. It's overloaded to operate on a wide range of value types: int == int, float == float, double == float, int == short, etc. You can overload the equality operator so that it can be used to compare your user-defined types with practically any other type of object.

Overloaded operators provide an elegant way to manipulate user-defined types. The decision regarding which operators should be overloaded to manipulate a particular class of objects is a function of your design. This chapter will help you understand how to overload C# operators and show you when it is appropriate to overload the different types of operators in the context of your design.

Although I will list all of the operators that can be overloaded in C#, I will not show you an example of how to overload every single operator. Many operators can be grouped together, like the binary arithmetic operators. Knowing how to overload one in the group leads to an understanding of how to overload the others. Most of the operators can be treated in this fashion.

When you get the hang of operator overloading and get used to thinking of when and how to incorporate overloaded operators in your class design, you will miss not being able to overload operators when programming in a language like Java. Overloaded operators, used in the right context, lead to cleaner, easier to read and understand code.

Operator Overloading

The C# language allows you to overload certain operators so that you can apply operator semantics to your user-defined types. Overloading an operator is simply the act of expanding the scope of the operator's defined behaviors so that the C# compiler understands how to apply the operator to your user-defined types. This gives you the ability to more naturally manipulate user-defined type objects in the context of your programs.

Now, you do not want, nor do you need, to overload operators just because you can. You must consider, as part of the design process, how you want your user-defined types to behave in a program. If overloading a particular operator facilitates the more natural manipulation of a user-defined type then doing so is a good design decision. You must also strive to preserve the spirit of the operator's intended semantics and guard against implementing unnatural behavior. This is easy to do if you stop for a moment to consider how a particular operator behaves when applied to C#'s predefined value or reference types.

Overloadable Operators

Table 21-1 lists the operators that can be overloaded in C#.

Operator Type	Operators	Notes
Unary	+,-,!,~,++,--,`true`, `false`	`true` and `false` must be overloaded in pairs.
Binary	+,-,*,/,%,&,|,^,<<,>>	
Comparison	==, !=, <,>,<=, >=	Must be overloaded in pairs.
Implicit cast	`implicit`†	
Explicit cast	`explicit`†	
† implicit and explicit are not operators. They are keywords used to declare implicit and explicit conversion operators		

Table 21-1: Overloadable Operators

© 2008 Rick Miller — All Rights Reserved

Referring to Table 21-1 — the comparison operators must be overloaded in pairs as do the `true` and `false` operators. This means that if you overload the `==` operator you must overload the `!=` operator as well. The keywords `implicit` and `explicit`, while not operators, are used to declare implicit and explicit cast operators.

Quick Review

Operator overloading allows you to more naturally manipulate user-defined type objects. As part of the design process, you must consider how you want your user-defined type objects to behave in a program. If overloading a particular operator facilitates the more natural manipulation of a user-defined type object then doing so is a good design decision. You must also strive to preserve the spirit of the operator's intended semantics and guard against implementing unnatural behavior.

Overloading Unary Operators

Unary operators operate on one argument. The general method signature for an overloaded unary operator is shown in Figure 21-1.

public static MyType operator + (MyType mt)

Figure 21-1: Method Signature for Overloaded Unary Operator

Referring to Figure 21-1 — overloaded operator methods are declared to be public and static and have a return type of the type in which they appear. In this particular example, the type name is MyType and the operator being overloaded is the unary + operator.

If you are overloading a logical operator the return type will be of type bool, as is shown in Figure 21-2.

public static bool operator ! (MyType mt)

Figure 21-2: Method Signature for Overloaded Unary Logical Operator

+,- Operators

The unary + operator, when applied to numeric types, returns the value of the operand. The unary – operator, when applied to numeric types, returns the numeric negation of the operand. How might these two operators be overloaded in the context of a user-defined type? Example 21.1 gives the code for a class named MyType, which includes both the overloaded + and – unary operators.

21.1 MyType.cs

```
1    using System;
2
3    public class MyType {
4      private int _intField;
5
6      public int IntField {
7        get { return _intField; }
8        set { _intField = value; }
9      }
10
11     public MyType():this(5){ }
12
13     public MyType(int intField){
14       _intField = intField;
15     }
16
17     public static MyType operator +(MyType mt){
18       mt.IntField = (+mt.IntField);
```


© 2008 Rick Miller — All Rights Reserved

```
19        return mt;
20      }
21
22      public static MyType operator -(MyType mt){
23        mt.IntField = (-mt.IntField);
24        return mt;
25      }
26    } // end class definition
```

Referring to Example 21.1 — the class MyType has one private field named _intField and one public property named IntField. It has a default constructor which sets the value of _intField to 5. Its second constructor sets the value of _intField to the value supplied via the constructor parameter. On line 17 the unary + operator is overloaded to work on objects of MyType. On line 18, the + operator is applied to the mt parameter's IntField property. Finally, on line 19, the mt parameter is returned.

The unary – operator is overloaded beginning on line 22. It looks exactly the same as the previous method, except that the unary – operator is applied to the mt parameter's IntField property.

Example 21.2 gives the code for a short application named MainApp that tests these newly overloaded operators.

21.2 MainApp.cs (+ and - operators)

```
1     using System;
2
3     public class MainApp {
4       public static void Main(){
5         MyType mt = new MyType();
6         Console.WriteLine(mt.IntField);
7         MyType mt2 = new MyType(-5);
8         Console.WriteLine(mt2.IntField);
9         mt2 = +mt2;
10        Console.WriteLine(mt2.IntField);
11        mt = -mt;
12        mt2 = -mt2;
13        Console.WriteLine(mt.IntField);
14        Console.WriteLine(mt2.IntField);
15      }
16    }
```

Referring to Example 21.2 — this short program creates two instances of MyType and initializes references mt and mt2. The value of mt.IntField is 5 (the default value), and the value of mt2.IntField is -5. The unary + and – operators are then applied to each of the objects and the results are printed to the console. Figure 21-3 shows the results of running this program.

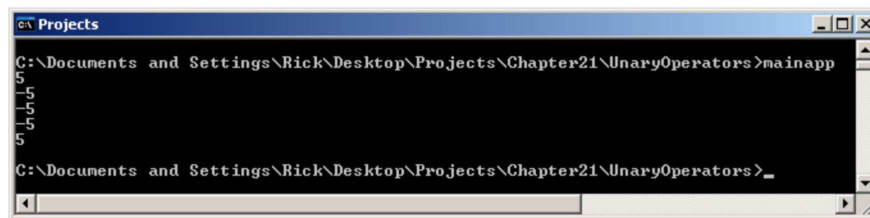


Figure 21-3: Results of Running Example 21.2

! Operator

The unary negation operator **!** is a logical operator that negates its operand. The overloaded **!** operator returns a boolean value. Example 21.3 shows how this operator might be overloaded in the context of MyType.

21.3 MyType.cs (! operator)

```
1     using System;
2
3     public class MyType {
4       private int _intField;
5
6       public int IntField {
7         get { return _intField; }
8         set { _intField = value; }
9       }
10
11      public MyType():this(5){ }
12
13      public MyType(int intField){
14        _intField = intField;
```

 © 2008 Rick Miller — All Rights Reserved

```
15     }
16
17     public static MyType operator +(MyType mt){
18       mt.IntField = (+mt.IntField);
19       return mt;
20     }
21
22     public static MyType operator -(MyType mt){
23       mt.IntField = (-mt.IntField);
24       return mt;
25     }
26
27     public static bool operator !(MyType mt){
28       bool retVal = true;
29       if(mt.IntField >= 0){
30         retVal = false;
31       }
32       return retVal;
33     }
34   } // end class definition
```

Referring to Example 21.3 — the overloaded `!` operator begins on line 27. The method returns *true* if the value of IntField is <= 0 and *false* otherwise. Example 21.4 shows the negation operator in action.

21.4 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        MyType mt = new MyType();
6        Console.WriteLine(mt.IntField);
7        mt = -mt;
8        Console.WriteLine(mt.IntField);
9        if(!mt){
10         Console.WriteLine(mt.IntField);
11       }
12     }
13   }
```

Referring to Example 21.4 — the ! operator is applied to the mt reference on line 9. The results of running this short program are shown in Figure 21-4.

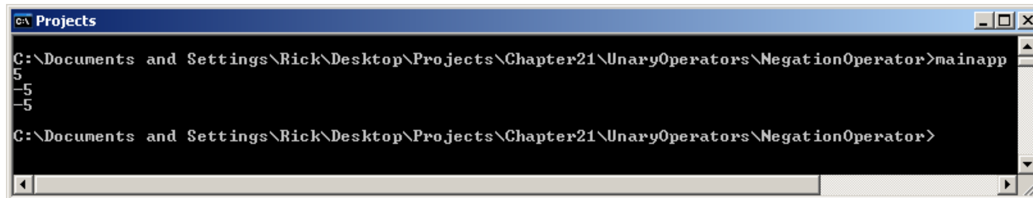


Figure 21-4: Results of Running Example 21.4

`true`, `false` Operators

The `true` and `false` operators must be overloaded in pairs. Overload these operators if you want to use your user defined type objects in conditional expressions. Example 21.5 gives the code for the modified MyType class with the overloaded `true` and `false` operators.

21.5 MyType.cs (true & false operators)

```
1    using System;
2
3    public class MyType {
4      private int _intField;
5
6      public int IntField {
7        get { return _intField; }
8        set { _intField = value; }
9      }
10
11     public MyType():this(5){
12     }
13
14     public MyType(int intField){
15       _intField = intField;
16     }
17
```

 © 2008 Rick Miller — All Rights Reserved

```
18      public static MyType operator +(MyType mt){
19        mt.IntField = (+mt.IntField);
20        return mt;
21      }
22
23      public static MyType operator -(MyType mt){
24        mt.IntField = (-mt.IntField);
25        return mt;
26      }
27
28      public static bool operator ! (MyType mt){
29        bool retVal = true;
30        if(mt.IntField >= 0){
31          retVal = false;
32        }
33        return retVal;
34      }
35
36      public static bool operator true(MyType mt){
37        bool retVal = true;
38        if(mt.IntField <= 0){
39          retVal = false;
40        }
41        return retVal;
42      }
43
44      public static bool operator false(MyType mt){
45       bool retVal = false;
46        if(mt.IntField > 0){
47          retVal = true;
48        }
49        return retVal;
50      }
51    } // end class definition
```

Referring to Example 21.5 — the overloaded `true` operator starts on line 36. It returns *true* if the value of IntField is greater than 0 and *false* otherwise. The overloaded `false` operator, which starts on line 44, behaves in the opposite fashion. Example 21.6 shows these two operators in action.

21.6 MainApp.cs

```
1     using System;
2
3     public class MainApp {
4       public static void Main(){
5         MyType mt = new MyType();
6         Console.WriteLine(mt.IntField);
7         if(mt){
8           Console.WriteLine("if statement evaluated to true. Value of IntField = " + mt.IntField);
9         }else{
10          Console.WriteLine("if statement evaluated to false. Value of IntField = " + mt.IntField);
11        }
12        mt = -mt;
13        if(mt){
14          Console.WriteLine("if statement evaluated to true. Value of IntField = " + mt.IntField);
15        }else{
16          Console.WriteLine("if statement evaluated to false. Value of IntField = " + mt.IntField);
17        }
18      }
19    }
```

Referring to Example 21.6 — this program creates an instance of MyType with a default IntField value of 5. The first `if` statement on line 7 will evaluate to *true*. On line 12, I negate the value of IntField with the overloaded `–` operator. The second `if` statement evaluates to *false*. Figure 21-5 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter21\UnaryOperators\truefalse>mainapp
5
if statement evaluated to false. Value of IntField = 5
if statement evaluated to true. Value of IntField = -5

C:\Documents and Settings\Rick\Desktop\Projects\Chapter21\UnaryOperators\truefalse>_
```

Figure 21-5: Results of Running Example 21.6

 © 2008 Rick Miller — All Rights Reserved

It seems, though, that since I have overloaded the `!` operator I can eliminate the repetitive code contained within the overloaded `true` and `false` operators. Example 21.7 offers an alternative implementation of the `true` and `false` operators coded with the help of the overloaded `!` operator.

21.7 MyType.cs (alternative true & false operator implementation)

```
1    using System;
2
3    public class MyType {
4      private int _intField;
5
6      public int IntField {
7        get { return _intField; }
8        set { _intField = value; }
9      }
10
11     public MyType():this(5){
12     }
13
14     public MyType(int intField){
15       _intField = intField;
16     }
17
18     public static MyType operator +(MyType mt){
19       mt.IntField = (+mt.IntField);
20       return mt;
21     }
22
23     public static MyType operator -(MyType mt){
24       mt.IntField = (-mt.IntField);
25       return mt;
26     }
27
28     public static bool operator ! (MyType mt){
29       bool retVal = true;
30       if(mt.IntField >= 0){
31         retVal = false;
32       }
33       return retVal;
34     }
35
36     public static bool operator true(MyType mt){
37       return !mt;
38     }
39
40     public static bool operator false(MyType mt){
41      return !mt;
42     }
43   } // end class definition
```

Referring to Example 21.7 — the overloaded `true` and `false` operators have been implemented with the help of the overloaded `!` operator. I'll leave it as an exercise for you to validate that this code behaves as expected.

++ --, Operators

The unary increment and decrement operators do not need to be overloaded in pairs, but if you do one you may as well do the other. Example 21.8 gives the modified MyType class showing how to overload these two operators.

21.8 MyType.cs (++ & -- operators)

```
1    using System;
2
3    public class MyType {
4      private int _intField;
5
6      public int IntField {
7        get { return _intField; }
8        set { _intField = value; }
9      }
10
11     public MyType():this(5){
12     }
13
14     public MyType(int intField){
15       _intField = intField;
16     }
17
18     public static MyType operator +(MyType mt){
19       mt.IntField = (+mt.IntField);
20       return mt;
```

 © 2008 Rick Miller — All Rights Reserved

```
21      }
22
23      public static MyType operator -(MyType mt){
24        mt.IntField = (-mt.IntField);
25        return mt;
26      }
27
28      public static bool operator ! (MyType mt){
29        bool retVal = true;
30        if(mt.IntField >= 0){
31          retVal = false;
32        }
33        return retVal;
34      }
35
36      public static bool operator true(MyType mt){
37        return !mt;
38      }
39
40      public static bool operator false(MyType mt){
41       return !mt;
42      }
43
44      public static MyType operator ++(MyType mt){
45        MyType result = new MyType(mt.IntField);
46        ++result.IntField;
47        return result;
48      }
49
50      public static MyType operator --(MyType mt){
51        MyType result = new MyType(mt.IntField);
52        --result.IntField;
53        return result;
54      }
55
56      public override String ToString(){
57        return IntField.ToString();
58      }
59    } // end class definition
```

Referring to Example 21.8 — the overloaded ++ and -- operators first create a new instance of MyType named result passing into the constructor the value of lhs.IntField. The result.IntField is incremented or decremented as required and the new reference returned. Note that I have also overridden the ToString() method to allow more natural use of MyType objects in the Console.WriteLine() method. Example 21.9 shows these overloaded operators in action.

21.9 MainApp.cs

```
1     using System;
2
3     public class MainApp {
4       public static void Main(){
5         MyType mt = new MyType();
6         Console.WriteLine(mt);
7         Console.WriteLine(mt++);
8         Console.WriteLine(++mt);
9         Console.WriteLine(--mt);
10        Console.WriteLine(mt--);
11      }
12    }
```

Figure 21-6 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter21\UnaryOperators\IncrementDecrement>mainapp
5
5
7
6
6

C:\Documents and Settings\Rick\Desktop\Projects\Chapter21\UnaryOperators\IncrementDecrement>_
```

Figure 21-6: Results of Running Example 21.9

Referring to Figure 21-6 — it looks like everything works fine. The semantics of the pre- and postfix increment and decrement operators have been preserved. By this I mean that when the ++ operator is applied in prefix fashion (*i.e.,* ++mt) the value of mt.IntField is incremented and then used in the expression. When applied in postfix fashion (*i.e.,* mt++) the value of mt.IntField is used in the expression and then incremented. The same holds true for the decrement operator.

 © 2008 Rick Miller — All Rights Reserved

Quick Review

Unary operators operate on one operand. Overloaded operator methods are declared to be public and static and have a return type of the type in which they appear. If you are overloading a logical operator the return type will be of type bool. You can overload the `true` and `false` operators with the help of the negation operator. When overloading the increment and decrement operator be sure to create and return a new instance of your type.

Overloading Binary Operators

Binary operators take two operands, a left-hand side and a right-hand side. The signature will differ depending on what other types you want your user-defined types to play nicely with. The signature for an overloaded binary + operator that works on two objects of type MyType is shown in Figure 21-7.

public static MyType operator +(MyType lhs, MyType rhs)

Figure 21-7: Overloaded Binary + Operator Signature that Operates on Two Objects of Type MyType

Figure 21-8 shows the signature of the same overloaded operator that works on objects of MyType and integer.

public static MyType operator +(MyType lhs, int rhs)

Figure 21-8: Overloaded Binary + Operator Signature that Operates on Objects of MyType and Integer

+, – Operators

When overloading the + and – operators for arithmetic types you must preserve the expected operator semantics. Example 21.10 shows how the + and – operators would be overloaded for the MyType class.

21.10 MyType.cs (+ and - operators)

```
1    using System;
2
3    public class MyType {
4      private int _intField;
5
6      public int IntField {
7        get { return _intField; }
8        set { _intField = value; }
9      }
10
11     public MyType():this(5){
12     }
13
14     public MyType(int intField){
15       _intField = intField;
16     }
17
18     public static MyType operator +(MyType mt){
19       mt.IntField = (+mt.IntField);
20       return mt;
21     }
22
23     public static MyType operator -(MyType mt){
24       mt.IntField = (-mt.IntField);
25       return mt;
26     }
27
28     public static bool operator ! (MyType mt){
29       bool retVal = true;
30       if(mt.IntField >= 0){
31         retVal = false;
32       }
```

 © 2008 Rick Miller — All Rights Reserved

```
33        return retVal;
34      }
35
36      public static bool operator true(MyType mt){
37        return !mt;
38      }
39
40      public static bool operator false(MyType mt){
41       return !mt;
42      }
43
44      public static MyType operator ++ (MyType mt){
45        MyType result = new MyType(mt.IntField);
46        ++result.IntField;
47        return result;
48      }
49
50      public static MyType operator -- (MyType mt){
51        MyType result = new MyType(mt.IntField);
52        --result.IntField;
53        return result;
54      }
55
56      public static MyType operator +(MyType lhs, MyType rhs){
57        MyType result = new MyType(lhs.IntField);
58        result.IntField += rhs.IntField;
59        return result;
60
61      }
62
63      public static MyType operator -(MyType lhs, MyType rhs){
64        MyType result = new MyType(lhs.IntField);
65        result.IntField -= rhs.IntField;
66        return result;
67      }
68
69      public static MyType operator +(MyType lhs, int rhs){
70        MyType result = new MyType(lhs.IntField);
71        result.IntField += rhs;
72        return result;
73      }
74
75      public static MyType operator -(MyType lhs, int rhs){
76        MyType result = new MyType(lhs.IntField);
77        result.IntField -= rhs;
78        return result;
79      }
80
81      public override String ToString(){
82        return IntField.ToString();
83      }
84    } // end class definition
```

Referring to Example 21.10 — I have overloaded the **+** and **–** operators twice: once to work on two objects of MyType and again to work on an object of MyType and an integer. Note how in each of the methods I first create a new instance of MyType, passing into its constructor the value of the lhs.IntField. I then add to it the value of the rhs.IntField, or simply the rhs in the case of integers. Example 21.11 shows these operators in action.

21.11 MainApp.cs

```
1     using System;
2
3     public class MainApp {
4       public static void Main(){
5         MyType mt = new MyType();
6         Console.WriteLine("mt = " + mt);
7         Console.WriteLine("Should be 7:" + (mt++ + 2));
8         Console.WriteLine("mt = " + mt);
9         Console.WriteLine("Should be 9:" + (++mt + 2));
10        Console.WriteLine("mt = " + mt);
11        Console.WriteLine("Should be 4:" + (--mt - 2));
12        Console.WriteLine("mt = " + mt);
13        Console.WriteLine("Should be 4:" + (mt-- - 2));
14        Console.WriteLine("mt = " + mt);
15        Console.WriteLine("----------------------------");
16        int i = 5;
17        Console.WriteLine("i = " + i);
18        Console.WriteLine("Should be 7:" + (i++ + 2));
19        Console.WriteLine("i = " + i);
20        Console.WriteLine("Should be 9:" + (++i + 2));
21        Console.WriteLine("i = " + i);
```

 © 2008 Rick Miller — All Rights Reserved

```
22        Console.WriteLine("Should be 4:" + (--i - 2));
23        Console.WriteLine("i = " + i);
24        Console.WriteLine("Should be 4:" + (i-- - 2));
25        Console.WriteLine("i = " + i);
26        Console.WriteLine("-----------------------------");
27        MyType mt2 = new MyType();
28        Console.WriteLine("Should be 10:" + (mt + mt2));
29        Console.WriteLine("mt = " + mt);
30        Console.WriteLine("mt2 = " + mt2);
31        Console.WriteLine("Should be 0:" + (mt - mt2));
32        Console.WriteLine("mt = " + mt);
33        Console.WriteLine("mt2 = " + mt2);
34      }
35    }
```

Referring to Example 21.11 — this program exercises both the **+** and **–** operators as well as the pre- and postfix increment and decrement operators. The code on lines 16 through 25 provides a reference output against which to compare our overloaded operator behavior. Figure 21-9 shows the results of running this program.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter21\BinaryOperators\PlusMinus>mainapp
mt = 5
Should be 7:7
mt = 6
Should be 9:9
mt = 7
Should be 4:4
mt = 6
Should be 4:4
mt = 5
----------------------------
i = 5
Should be 7:7
i = 6
Should be 9:9
i = 7
Should be 4:4
i = 6
Should be 4:4
i = 5
-----------------------------
Should be 10:10
mt = 5
mt2 = 5
Should be 0:0
mt = 5
mt2 = 5

C:\Documents and Settings\Rick\Desktop\Projects\Chapter21\BinaryOperators\PlusMinus>_
```

Figure 21-9: Results of Running Example 21.11

*, / Operators

The multiplication and division operators are implemented very much like the addition and subtraction operators. Example 21.12 gives the code for the MyType class showing how the `*` and `/` operators can be overloaded to operate on objects of MyType and integer.

21.12 MyType.cs (and / operators)*

```
1     using System;
2
3     public class MyType {
4       private int _intField;
5
6       public int IntField {
7         get { return _intField; }
8         set { _intField = value; }
9       }
10
11      public MyType():this(5){
12      }
13
14      public MyType(int intField){
15        _intField = intField;
16      }
17
18      public static MyType operator +(MyType mt){
19        mt.IntField = (+mt.IntField);
20        return mt;
21      }
22
```

 © 2008 Rick Miller — All Rights Reserved

```
23      public static MyType operator -(MyType mt){
24        mt.IntField = (-mt.IntField);
25        return mt;
26      }
27
28      public static bool operator ! (MyType mt){
29        bool retVal = true;
30        if(mt.IntField >= 0){
31          retVal = false;
32        }
33        return retVal;
34      }
35
36      public static bool operator true(MyType mt){
37        return !mt;
38      }
39
40      public static bool operator false(MyType mt){
41       return !mt;
42      }
43
44      public static MyType operator ++ (MyType mt){
45        MyType result = new MyType(mt.IntField);
46        ++result.IntField;
47        return result;
48      }
49
50      public static MyType operator -- (MyType mt){
51        MyType result = new MyType(mt.IntField);
52        --result.IntField;
53        return result;
54      }
55
56      public static MyType operator +(MyType lhs, MyType rhs){
57        MyType result = new MyType(lhs.IntField);
58        result.IntField += rhs.IntField;
59        return result;
60
61      }
62
63      public static MyType operator -(MyType lhs, MyType rhs){
64        MyType result = new MyType(lhs.IntField);
65        result.IntField -= rhs.IntField;
66        return result;
67      }
68
69      public static MyType operator +(MyType lhs, int rhs){
70        MyType result = new MyType(lhs.IntField);
71        result.IntField += rhs;
72        return result;
73      }
74
75      public static MyType operator -(MyType lhs, int rhs){
76        MyType result = new MyType(lhs.IntField);
77        result.IntField -= rhs;
78        return result;
79      }
80
81      public static MyType operator *(MyType lhs, MyType rhs){
82        MyType result = new MyType(lhs.IntField);
83        result.IntField *= rhs.IntField;
84        return result;
85      }
86
87      public static MyType operator *(MyType lhs, int rhs){
88        MyType result = new MyType(lhs.IntField);
89        result.IntField *= rhs;
90        return result;
91      }
92
93      public static MyType operator /(MyType lhs, MyType rhs){
94        MyType result = new MyType(lhs.IntField);
95        result.IntField /= rhs.IntField;
96        return result;
97      }
98
99      public static MyType operator /(MyType lhs, int rhs){
100       MyType result = new MyType(lhs.IntField);
101       result.IntField /= rhs;
102       return result;
103     }
```


© 2008 Rick Miller — All Rights Reserved

```
104
105    public override String ToString(){
106      return IntField.ToString();
107    }
108  } // end class definition
```

Referring to Example 21.12 — the multiplication and division operators are overloaded to operate on either two MyType objects or one MyType object and an integer. The implementation of these methods follows that of the addition and subtraction operators. Example 21.13 shows these operators in action.

21.13 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        MyType mt = new MyType();
6        MyType mt2 = new MyType();
7        Console.WriteLine("mt = " + mt);
8        Console.WriteLine("mt2 = " + mt2);
9        Console.WriteLine("Should be 25: " + (mt * mt2));
10       Console.WriteLine("mt = " + mt);
11       Console.WriteLine("mt2 = " + mt2);
12       Console.WriteLine("Should be 1: " + (mt / mt2));
13       Console.WriteLine("mt = " + mt);
14       Console.WriteLine("mt2 = " + mt2);
15       Console.WriteLine("Should be 25: " + (mt * 5));
16       Console.WriteLine("mt = " + mt);
17       Console.WriteLine("Should be 1: " + (mt / mt2));
18       Console.WriteLine("mt = " + mt);
19     }
20   }
```

Figure 21-10 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter21\BinaryOperators\MultiplyDivide>mainapp
mt = 5
mt2 = 5
Should be 25: 25
mt = 5
mt2 = 5
Should be 1: 1
mt = 5
mt2 = 5
Should be 25: 25
mt = 5
Should be 1: 1
mt = 5

C:\Documents and Settings\Rick\Desktop\Projects\Chapter21\BinaryOperators\MultiplyDivide>_
```

Figure 21-10: Results of Running Example 21.13

&, | Operators

These two operators behave differently when applied to bool and integral types. When applied to bool types the & operator computes the *logical AND* operation but compares the values of both operands even if the first operand evaluates to false. When applied to integral operands, the & operator performs a *bitwise AND* operation. The | operator performs a *logical OR* operation on bool operands and a *bitwise OR* operation on integral operands.

I think I'll go with the bitwise path for the MyType class. Example 21.14 gives the code for the modified MyType class showing these two overloaded operators.

21.14 MyType.cs (bitwise & and |)

```
1    using System;
2
3    public class MyType {
4      private int _intField;
5
6      public int IntField {
7        get { return _intField; }
8        set { _intField = value; }
9      }
10
11     public MyType():this(5){
12     }
13
14     public MyType(int intField){
15       _intField = intField;
```

 © 2008 Rick Miller — All Rights Reserved

```
16      }
17
18      public static MyType operator +(MyType mt){
19        mt.IntField = (+mt.IntField);
20        return mt;
21      }
22
23      public static MyType operator -(MyType mt){
24        mt.IntField = (-mt.IntField);
25        return mt;
26      }
27
28      public static bool operator ! (MyType mt){
29        bool retVal = true;
30        if(mt.IntField >= 0){
31          retVal = false;
32        }
33        return retVal;
34      }
35
36      public static bool operator true(MyType mt){
37        return !mt;
38      }
39
40      public static bool operator false(MyType mt){
41       return !mt;
42      }
43
44      public static MyType operator ++ (MyType mt){
45        MyType result = new MyType(mt.IntField);
46        ++result.IntField;
47        return result;
48      }
49
50      public static MyType operator -- (MyType mt){
51        MyType result = new MyType(mt.IntField);
52        --result.IntField;
53        return result;
54      }
55
56      public static MyType operator +(MyType lhs, MyType rhs){
57        MyType result = new MyType(lhs.IntField);
58        result.IntField += rhs.IntField;
59        return result;
60
61      }
62
63      public static MyType operator -(MyType lhs, MyType rhs){
64        MyType result = new MyType(lhs.IntField);
65        result.IntField -= rhs.IntField;
66        return result;
67      }
68
69      public static MyType operator +(MyType lhs, int rhs){
70        MyType result = new MyType(lhs.IntField);
71        result.IntField += rhs;
72        return result;
73      }
74
75      public static MyType operator -(MyType lhs, int rhs){
76        MyType result = new MyType(lhs.IntField);
77        result.IntField -= rhs;
78        return result;
79      }
80
81      public static MyType operator *(MyType lhs, MyType rhs){
82        MyType result = new MyType(lhs.IntField);
83        result.IntField *= rhs.IntField;
84        return result;
85      }
86
87      public static MyType operator *(MyType lhs, int rhs){
88        MyType result = new MyType(lhs.IntField);
89        result.IntField *= rhs;
90        return result;
91      }
92
93      public static MyType operator /(MyType lhs, MyType rhs){
94        MyType result = new MyType(lhs.IntField);
95        result.IntField /= rhs.IntField;
96        return result;
```


© 2008 Rick Miller — All Rights Reserved

```
97     }
98
99     public static MyType operator /(MyType lhs, int rhs){
100      MyType result = new MyType(lhs.IntField);
101      result.IntField /= rhs;
102      return result;
103    }
104
105    public static MyType operator &(MyType lhs, MyType rhs){
106      MyType result = new MyType(lhs.IntField);
107      result.IntField &= rhs.IntField;
108      return result;
109    }
110
111    public static MyType operator |(MyType lhs, MyType rhs){
112      MyType result = new MyType(lhs.IntField);
113      result.IntField |= rhs.IntField;
114      return result;
115    }
116
117    public override String ToString(){
118      return IntField.ToString();
119    }
120  } // end class definition
```

Referring to Example 21.14 — the bitwise & operator creates a new instance of MyType passing into its constructor the value of lhs.IntField. It then sets result.IntField to the bitwise AND of itself and rhs.IntField. I've overloaded these operators to use either two MyType objects or one MyType object and an integer. Example 21.15 shows these overloaded bitwise operators in action.

21.15 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        MyType mt = new MyType();
6        MyType mt2 = new MyType(6);
7        Console.WriteLine("Should be 4:" + (mt & mt2)); // 0101 & 0110 = 0100
8        Console.WriteLine("Should be 7:" + (mt | mt2)); // 0101 | 0110 = 0111
9      }
10   }
```

Figure 21.11 shows the results of running this program.

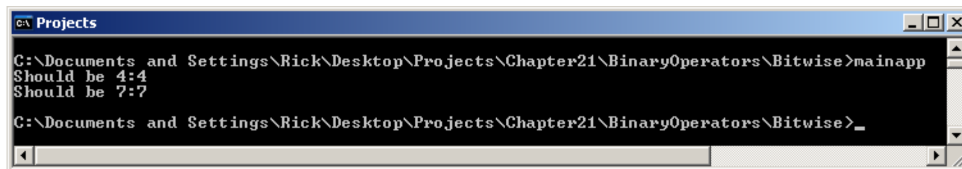


Figure 21-11: Results of Running Example 21.15

Quick Review

Binary operators operate on two operands. You can overload binary operators to operate on two operands of the same user-defined type, or on one user-defined type and any other type, which may be another user-defined type.

Overloading Comparison Operators

The comparison operators must be overloaded in pairs. That is, if you overload the equality operator ==, you must also overload the inequality operator !=. The same holds true for the less-than and greater-than operators <,>, and the less-than-or-equal-to and greater-than-or-equal-to operators <=, >=.

 © 2008 Rick Miller — All Rights Reserved

==, != , < , > , <= , >= Operators

The overloaded method signature for the comparison equality operator == is shown in Figure 21-12.

public static bool operator ==(MyType lhs, MyType rhs)

Figure 21-12: Method Signature for Overloaded Equality Operator

Referring to Figure 21-12 — in this example, the equality operator is overloaded to work on two MyType objects. If you wanted to compare a MyType object to another type you would make the appropriate change to the type of the rhs parameter.

Note: You usually don't need to overload the equality and inequality operators == and != for reference (*i.e.*, class) types because their default behavior suffices to settle the matter. In other words, if I compare two objects of the same type and their addresses are the same they must be equal because they are the same object. On the other hand, if you are comparing two distinct objects, the inequality operator will return true, regardless of whether or not the objects are equal in all other aspects. So, overloading the == and != operators requires more thought regarding how you want to manipulate your user-defined objects. Also, if you overload the == and != operators you should also override the Object.Equals(Obejct o) and Object.GetHashCode() methods. (This topic is covered in more detail in *Chapter 22: Well-Behaved Objects*.)

Example 21.16 shows how the comparison operators would be overloaded in the context of MyType

21.16 MyType.cs (comparison operators)

```
1    using System;
2
3    public class MyType {
4      private int _intField;
5
6      public int IntField {
7        get { return _intField; }
8        set { _intField = value; }
9      }
10
11     public MyType():this(5){
12     }
13
14     public MyType(int intField){
15       _intField = intField;
16     }
17
18     public static MyType operator +(MyType mt){
19       mt.IntField = (+mt.IntField);
20       return mt;
21     }
22
23     public static MyType operator -(MyType mt){
24       mt.IntField = (-mt.IntField);
25       return mt;
26     }
27
28     public static bool operator ! (MyType mt){
29       bool retVal = true;
30       if(mt.IntField >= 0){
31         retVal = false;
32       }
33       return retVal;
34     }
35
36     public static bool operator true(MyType mt){
37       return !mt;
38     }
39
40     public static bool operator false(MyType mt){
41      return !mt;
42     }
43
44     public static MyType operator ++ (MyType mt){
45       MyType result = new MyType(mt.IntField);
46       ++result.IntField;
47       return result;
48     }
49
```


© 2008 Rick Miller — All Rights Reserved

```
50      public static MyType operator -- (MyType mt){
51        MyType result = new MyType(mt.IntField);
52        --result.IntField;
53        return result;
54      }
55
56      public static MyType operator +(MyType lhs, MyType rhs){
57        MyType result = new MyType(lhs.IntField);
58        result.IntField += rhs.IntField;
59        return result;
60
61      }
62
63      public static MyType operator -(MyType lhs, MyType rhs){
64        MyType result = new MyType(lhs.IntField);
65        result.IntField -= rhs.IntField;
66        return result;
67      }
68
69      public static MyType operator +(MyType lhs, int rhs){
70        MyType result = new MyType(lhs.IntField);
71        result.IntField += rhs;
72        return result;
73      }
74
75      public static MyType operator -(MyType lhs, int rhs){
76        MyType result = new MyType(lhs.IntField);
77        result.IntField -= rhs;
78        return result;
79      }
80
81      public static MyType operator *(MyType lhs, MyType rhs){
82        MyType result = new MyType(lhs.IntField);
83        result.IntField *= rhs.IntField;
84        return result;
85      }
86
87      public static MyType operator *(MyType lhs, int rhs){
88        MyType result = new MyType(lhs.IntField);
89        result.IntField *= rhs;
90        return result;
91      }
92
93      public static MyType operator /(MyType lhs, MyType rhs){
94        MyType result = new MyType(lhs.IntField);
95        result.IntField /= rhs.IntField;
96        return result;
97      }
98
99      public static MyType operator /(MyType lhs, int rhs){
100       MyType result = new MyType(lhs.IntField);
101       result.IntField /= rhs;
102       return result;
103     }
104
105     public static MyType operator &(MyType lhs, MyType rhs){
106       MyType result = new MyType(lhs.IntField);
107       result.IntField &= rhs.IntField;
108       return result;
109     }
110
111     public static MyType operator |(MyType lhs, MyType rhs){
112       MyType result = new MyType(lhs.IntField);
113       result.IntField |= rhs.IntField;
114       return result;
115     }
116
117     public static bool operator ==(MyType lhs, MyType rhs){
118       return lhs.IntField == rhs.IntField;
119     }
120
121     public static bool operator !=(MyType lhs, MyType rhs){
122       return lhs.IntField != rhs.IntField;
123     }
124
125     public static bool operator <(MyType lhs, MyType rhs){
126       return lhs.IntField < rhs.IntField;
127     }
128
129     public static bool operator >(MyType lhs, MyType rhs){
130       return lhs.IntField > rhs.IntField;
```

 © 2008 Rick Miller — All Rights Reserved

```
131    }
132
133    public static bool operator <=(MyType lhs, MyType rhs){
134      return lhs.IntField <= rhs.IntField;
135    }
136
137    public static bool operator >=(MyType lhs, MyType rhs){
138      return lhs.IntField >= rhs.IntField;
139    }
140
141    public override String ToString(){
142      return IntField.ToString();
143    }
144  } // end class definition
```

Referring to Example 21.16 — the overloaded comparison operators begin on line 117 with the equality operator. Note that in this example it all boils down to comparing the lhs.IntField against the rhs.IntField and returning the result. Example 21.17 shows these operators in action.

21.17 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        MyType mt = new MyType();
6        MyType mt2 = new MyType(6);
7        Console.WriteLine("Should be True:" + (mt == mt));
8        Console.WriteLine("Should be False:" + (mt != mt));
9        Console.WriteLine("Should be True:" + (mt != mt2));
10       Console.WriteLine("should be False:" + (mt == mt2));
11       Console.WriteLine("Should be False:" + (mt > mt2));
12       Console.WriteLine("Should be False:" + (mt >= mt2));
13       Console.WriteLine("Should be True:" + (mt < mt2));
14       Console.WriteLine("Should be True:" + (mt <= mt2));
15     }
16   }
```

Alright, when you compile these files you'll receive several compiler warnings as are shown in Figure 21-13.

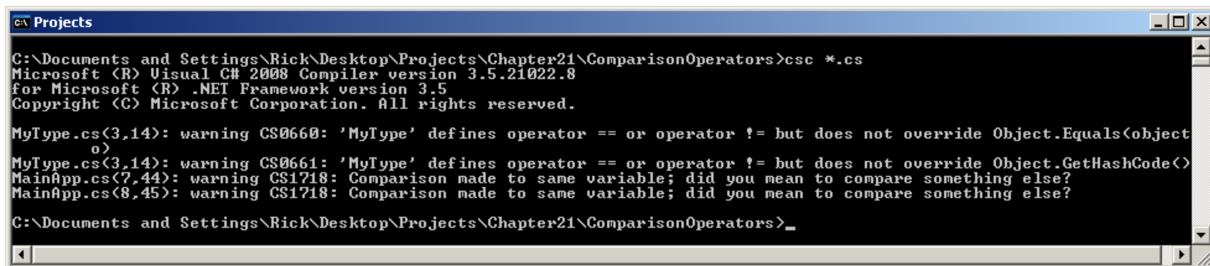


Figure 21-13: Compiler Warning — == and != Operators Need Special Attention

Referring to Figure 21-13 — the first two compiler warnings suggest that if you overload the equality and inequality operators **==** and **!=** then you might want to consider overriding the Object.Equals(Object o) and Object.GetHashCode() methods. The second two compiler warnings simply draw your attention to the comparison of the same variable. For the purposes of this chapter, you can safely ignore these warnings. Figure 21-14 shows the results of exercising the overloaded comparison operators.

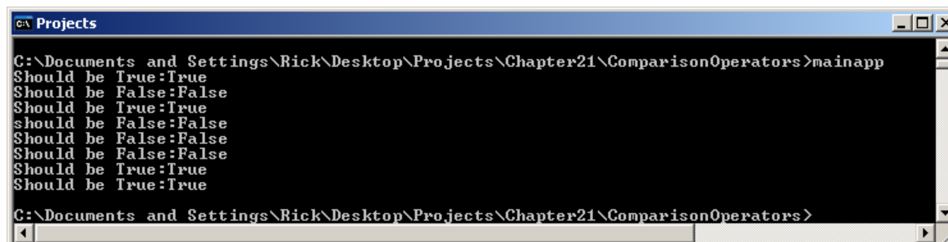


Figure 21-14: Results of Running Example 21.17

 © 2008 Rick Miller — All Rights Reserved

Quick Review

The comparison operators must be overloaded in pairs. If you overload the equality and inequality operators you should also override the Object.Equals(Object o) and the Object.GetHashCode() methods.

Creating Implicit And Explicit Cast Operators

If you would like to change a user-defined type object into another type object, you must implement a cast operator of which there are two kinds: `implicit` and `explicit`. Figure 21-15 shows the method signature of both an `implicit` and `explicit` cast operator.

public static implicit operator int(MyType mt)

public static explicit operator int(MyType mt)

Figure 21-15: Method Signatures for Implicit and Explicit Cast Operators

Referring to Figure 21-15 — each of these cast operators will allow the MyType objects to be cast to integers. If you wanted to cast MyType objects to anything other than an integer, you would need to provide an overloaded cast operator for each type you want to cast to.

Implicit vs. Explicit Cast

I put this section in just to remind you of the difference between an implicit and an explicit cast. With an implicit cast, you can simply assign the value of one type to another. The conversion between types is made automatically and without fuss. The following code snippet shows how an object of MyType would be implicitly cast to an integer:

```
MyType mt = new MyType(); // mt.IntField == 5 by default
int i = mt; // i is now equal to 5
```

An explicit cast requires the use of parentheses. You generally need to use an explicit cast when a conversion from one type to another has some type of ramification that must be considered, like when you cast from double to int, which most certainly will result in a loss of precision. The following code snippet shows how you would cast a double to an int:

```
Double d = 100.24;
int i = (int)d; // required because you will lose precision as a result
```

Overloaded Cast Operators Example

You can only overload one cast operator at a time, not both. Example 21.18 shows how the `explicit` cast operator would be implemented to allow explicit casts from MyType object to integers.

21.18 MyType.cs (explicit cast operator)

```
1    using System;
2
3    public class MyType {
4      private int _intField;
5
6      public int IntField {
7        get { return _intField; }
8        set { _intField = value; }
9      }
10
11     public MyType():this(5){
12     }
13
14     public MyType(int intField){
15       _intField = intField;
16     }
17
18     public static MyType operator +(MyType mt){
19       mt.IntField = (+mt.IntField);
```


© 2008 Rick Miller — All Rights Reserved

```
20        return mt;
21      }
22
23      public static MyType operator -(MyType mt){
24        mt.IntField = (-mt.IntField);
25        return mt;
26      }
27
28      public static bool operator ! (MyType mt){
29        bool retVal = true;
30        if(mt.IntField >= 0){
31          retVal = false;
32        }
33        return retVal;
34      }
35
36      public static bool operator true(MyType mt){
37        return !mt;
38      }
39
40      public static bool operator false(MyType mt){
41       return !mt;
42      }
43
44      public static MyType operator ++ (MyType mt){
45        MyType result = new MyType(mt.IntField);
46        ++result.IntField;
47        return result;
48      }
49
50      public static MyType operator -- (MyType mt){
51        MyType result = new MyType(mt.IntField);
52        --result.IntField;
53        return result;
54      }
55
56      public static MyType operator +(MyType lhs, MyType rhs){
57        MyType result = new MyType(lhs.IntField);
58        result.IntField += rhs.IntField;
59        return result;
60
61      }
62
63      public static MyType operator -(MyType lhs, MyType rhs){
64        MyType result = new MyType(lhs.IntField);
65        result.IntField -= rhs.IntField;
66        return result;
67      }
68
69      public static MyType operator +(MyType lhs, int rhs){
70        MyType result = new MyType(lhs.IntField);
71        result.IntField += rhs;
72        return result;
73      }
74
75      public static MyType operator -(MyType lhs, int rhs){
76        MyType result = new MyType(lhs.IntField);
77        result.IntField -= rhs;
78        return result;
79      }
80
81      public static MyType operator *(MyType lhs, MyType rhs){
82        MyType result = new MyType(lhs.IntField);
83        result.IntField *= rhs.IntField;
84        return result;
85      }
86
87      public static MyType operator *(MyType lhs, int rhs){
88        MyType result = new MyType(lhs.IntField);
89        result.IntField *= rhs;
90        return result;
91      }
92
93      public static MyType operator /(MyType lhs, MyType rhs){
94        MyType result = new MyType(lhs.IntField);
95        result.IntField /= rhs.IntField;
96        return result;
97      }
98
99      public static MyType operator /(MyType lhs, int rhs){
100       MyType result = new MyType(lhs.IntField);
```

 © 2008 Rick Miller — All Rights Reserved

```
101      result.IntField /= rhs;
102      return result;
103    }
104
105    public static MyType operator &(MyType lhs, MyType rhs){
106      MyType result = new MyType(lhs.IntField);
107      result.IntField &= rhs.IntField;
108      return result;
109    }
110
111    public static MyType operator |(MyType lhs, MyType rhs){
112      MyType result = new MyType(lhs.IntField);
113      result.IntField |= rhs.IntField;
114      return result;
115    }
116
117    public static bool operator ==(MyType lhs, MyType rhs){
118      return lhs.IntField == rhs.IntField;
119    }
120
121    public static bool operator !=(MyType lhs, MyType rhs){
122      return lhs.IntField != rhs.IntField;
123    }
124
125    public static bool operator <(MyType lhs, MyType rhs){
126      return lhs.IntField < rhs.IntField;
127    }
128
129    public static bool operator >(MyType lhs, MyType rhs){
130      return lhs.IntField > rhs.IntField;
131    }
132
133    public static bool operator <=(MyType lhs, MyType rhs){
134      return lhs.IntField <= rhs.IntField;
135    }
136
137    public static bool operator >=(MyType lhs, MyType rhs){
138      return lhs.IntField >= rhs.IntField;
139    }
140
141    public static explicit operator int(MyType mt){
142      return mt.IntField;
143    }
144
145    public override String ToString(){
146      return IntField.ToString();
147    }
148  } // end class definition
```

Referring to Example 21.18 — the `explicit` cast operator begins on line 141. Note that the return type is implied by the type of object you want to cast to, which in this case is an integer. Since the MyType.IntField property is already an integer, all you need to do is simply return its value as is done here.

Example 21.19 shows the explicit cast operator in action.

21.19 MainApp.cs

```
1   using System;
2
3   public class MainApp {
4     public static void Main(){
5       MyType mt = new MyType();
6       int i = (int)mt; // explicit cast
7       Console.WriteLine("i should be 5:" + i);
8     }
9   }
```

Figure 21-16 shows the results of running this program.

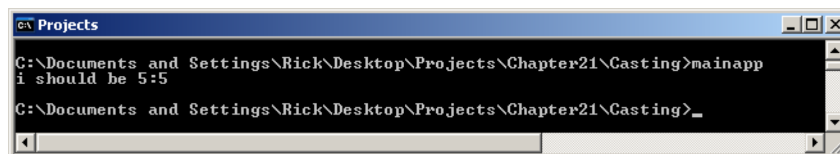


Figure 21-16: Results of Running Example 21.19

© 2008 Rick Miller — All Rights Reserved

Quick Review

You can implement either the implicit cast or the explicit cast, but not both.

The Assignment Operators: Things You Get For Free

You cannot explicitly overload the assignment operators. These include +=, -=, *=, /=, etc. However, if you overload the binary + operator, you get the += for free! The same holds true for the other assignment operators. Example 21.20 demonstrates the use of these operators on MyType and integer objects.

21.20 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        MyType mt = new MyType();
6        MyType mt2 = new MyType(6);
7        Console.WriteLine("mt = " + mt);
8        Console.WriteLine("mt2 = " + mt2);
9        MyType mt3 = mt2; // assignment
10       Console.WriteLine("mt3 should be 6:" + mt3);
11       mt += mt2; // addition assignment
12       Console.WriteLine("mt = " + mt);
13       mt *= mt;  // multiplication assignment
14       Console.WriteLine("mt = " + mt);
15       mt /= 2; // division assignment - will lose the remainder when doing integer division
16       Console.WriteLine("mt = " + mt);
17     }
18   }
```

Figure 21-17 shows the results of running this program.

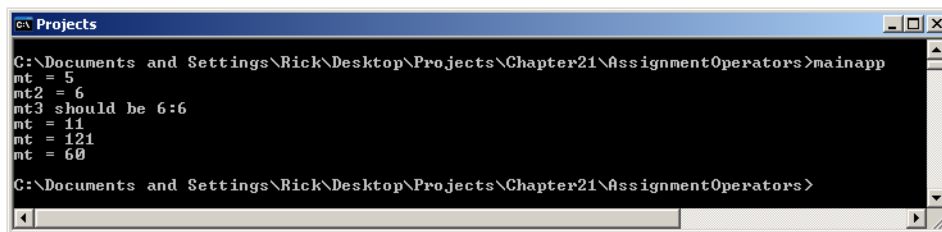


Figure 21-17: Results of Running Example 21.20

Quick Review

When you overload the binary arithmetic operators you get the assignment operators for free.

Summary

Operator overloading allows you to more natually manipulate user-defined type objects. You must consider, as part of the design process, how you want your user-defined types to behave in a program. If overloading a particular operator facilitates the more natural manipulation of a user-defined type object then doing so is a good design decision. You must also strive to preserve the spirit of the operator's intended semantics and guard against implementing unnatural behavior.

Unary operators operate on one operand. Overloaded operator methods are declared to be public and static and have a return type of the type in which they appear. If you are overloading a logical operator, the return type will be of type bool. You can overload the `true` and `false` operators with the help of the negation operator. When overloading the increment and decrement operators be sure to create and return a new instance of your type.

 © 2008 Rick Miller — All Rights Reserved

Binary operators operate on two operands. You can overload binary operators to operate on two operands of the same user-defined type, or on one user-defined type and any other type, which may be another user-defined type.

The comparison operators must be overloaded in pairs. If you overload the equality and inequality operators you may also want to override the Object.Equals(Object o) and the Object.GetHashCode() methods.

You can implement either the `implicit` cast or the `explicit` cast, but not both.

When you overload the binary arithmetic operators you get the assignment operators for free.

Skill-Building Exercises

1. **Programming Drill:** Compile and run the examples presented in this chapter.

2. **API Drill:** Visit the MSDN website and research any operators I have failed to cover in this chapter.

3. **Programming Drill:** Verify that the overloaded `true` and `false` operators implemented in Example 21.7 work as expected.

Suggested Projects

1. **Person Class Operators:** Overload the comparison operators for the Person class as given in Chapter 20. Write a short application to test your new operators. (**Hint:** Pick some attribute, maybe the Age property, with which you can compare different Person objects.)

Self-Test Questions

1. What's the purpose of operator overloading?

2. What should you strive to do when overloading operators for your user-defined types?

3. Unary operators operate on how many operands?

4. Binary operators operate on how many operands?

5. (T/F) Comparison operators must be overloaded in pairs.

6. (T/F) The `true` and `false` operators must be overloaded as a pair.

7. If you overload the binary arithmetic operator `+`, what assignment operator do you get for free?

8. How should you implement the increment and decrement operators?

9. Describe in your own words why it's a good idea to preserve the expected operator semantics when overloading operators for your user-defined types?

10. (T/F) You can overload both cast operators at the same time.

© 2008 Rick Miller — All Rights Reserved

References

Microsoft Developer Network (MSDN) *.NET Framework 3.0 and3.5 Reference Documentation* [www.msdn.com]

Notes

© 2008 Rick Miller — All Rights Reserved

Chapter 22

Contax T / Kodax Tri-X

Mayan Dogs, Yucatan, Mexico

Well-Behaved Objects

Learning Objectives

- List and describe the desirable characteristics of a well-behaved object
- List and describe the four categories of object behavior
- List and describe the seven object usage scenarios
- State the purpose of a copy constructor
- Describe the default behavior of the Object.Equals() method on value types
- Describe the default behavior of the Object.Equals() method on reference types
- List the rules associated with overriding the Object.Equals() method
- Explain why you must override Object.Equals() when overloading the equality operator
- State the characteristics of a good Object.GetHashCode() method
- Explain the relationship between the Object.Equals() and Object.GetHashCode() methods
- Explain the difference between a deep copy vs. a shallow copy
- Identify an object's natural ordering
- Demonstrate your ability to override Object methods to gain predictable user-defined object behavior

© 2008 Rick Miller — All Rights Reserved

Introduction

When designing user-defined data types, whether they be classes or structures, you must always ask yourself, "How should objects of this type behave in a program?" You must then takes steps to implement the user-defined type in a way that ensures its objects behave as expected.

Object behavior goes beyond an object's public interface. When considering how an object behaves, you must think about what happens when you create an object from scratch vs. creating an object from an existing object of the same or different type. This leads to a need to understand the difference between a deep copy vs. a shallow copy. A failure to understand this fundamental concept can lead to dire consequences and unexplained or irrational object behavior.

Another important characteristic of object behavior has to do with object equality. What does it mean to compare one object to another object of the same type? Do you accept the default behavior supplied by the Object.Equals() method or do you override it? What else must you do when you override Object.Equals()? What's the relationship between the Object.Equals() and the Object.GetHashCode() methods? You must be keenly aware of these and other object equality issues in order to make your objects behave properly.

If you intend to use user-defined objects in ordered collections such as trees, you must implement either IComparable<T> or an appropriate Comparer<T> object. To do this, you must understand what is meant by *natural ordering* when the term is used in the context of your user-defined types.

To make the subject easier to remember, I have categorized object behavior into four distinct groups: *fundamental behavior*, *copy/assignment behavior*, *equality behavior*, and *comparison/ordering behavior*. Some of what I discuss in this chapter has been covered earlier in the book, but most is new material. What I've tried to do here is present these issues as a coherent whole. I have also provided you with a handy check-off list of the seven object usage scenarios.

When you finish this chapter, you'll have a good understanding of these and other object behavior issues. The bottom line — getting your objects to behave properly in demanding situations.

Object Behavior Defined

"How should objects of this type behave in a program?" This question should be foremost on your mind when creating user-defined data types. To help you sort out the answer to this question, I have categorized object behavior into four groups: *fundamental behavior*, *copy/assignment behavior*, *equality behavior*, and *comparison/ordering behavior*. This section briefly describes each category. The remainder of the chapter covers each category in detail.

Fundamental Behavior

Fundamental behaviors include object creation (constructors), overriding the Object.ToString() method, object serialization, and member accessibility. By now you should be familiar with, and understand the purpose of, constructors. You should also by now know how to override the Object.ToString() method and understand when doing so is appropriate, so I won't dwell too long on these two topics. I will, however, go into more detail about object serialization and show you how to better control the serialization process. I will also summarize accessibility issues and show you how to apply the keywords `public`, `protected` and `private`.

Copy/Assignment Behavior

Constructors, as you know, are special methods that are used to properly initialize an object when it's created in memory. You can also create what are referred to as *copy constructors* to create objects from existing objects. In this section, I show you how to create copy constructors, and also show you how to implement the ICloneable interface. Another important topic covered in this section is a discussion of *deep copy* vs. *shallow copy*.

© 2008 Rick Miller — All Rights Reserved

Equality Behavior

If user-defined type objects are to be compared against each other for equality then you need to properly implement equality behavior. To do this, you need to understand how *reference equality* differs from *value equality*, how to override the Object.Equals() and Object.GetHashCode() methods, and how to implement a good custom hashcode algorithm.

Comparison/Ordering Behavior

Objects used in comparison and ordering operations must properly implement either the IComparable<T> interface or create Comparer<T> objects. Also in this section, I'll discuss the meaning of the term *natural ordering*.

Seven Object Usage Scenarios

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 22-1.

Yes/No	Usage Scenario	Implementation Requirement
	Will objects of this type be required to provide a string representation of themselves?	Override the Object.ToString() method. (Although not a strict requirement, it is a generally accepted practice to override the ToString() method in all cases.)
	Will objects of this type be compared against each other for equality?	Override the Object.Equals() method obeying the general contract specified in the .NET Framework documentation. Also override the Object.GetHashCode() method.
	Will objects of this type be inserted into a hash-based collection?	Override the Object.GetHashCode() method obeying the general contract specified in the .NET Framework documentation. Also override the Object.Equals() method.
	Will objects of this type be copied or cloned?	Use the Object.MemberwiseClone() method for simple value types. The MemberwiseClone() method performs a shallow copy. If the object being cloned contains other objects or collections of objects, you can create copy constructors or implement the ICloneable interface.
	Do objects of this type have a natural ordering? i.e., Will they be sorted? and Do you own the source code?	Implement the IComparable interface.
	Do objects of this type have different possible orderings? (Or, you don't own the source code.)	Create a separate comparer object by extending System.Collections.Generic.Comparer<T>.
	Will objects of this type be saved to disk or sent via a network?	Apply the Serializable attribute and, if necessary, implement custom serialization.

Table 22-1: Object Usage Scenario Evaluation Checklist

Referring to Table 22-1 — not all usage scenarios will apply to all classes. However, it is generally considered good 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.GetHashCode() methods are overridden together. This is because the correct behavior of one affects the behavior of the other. These two methods also bring up the notion of a *behavior contract*. A behavior 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.

© 2008 Rick Miller — All Rights Reserved

You'll find method behavior contract rules in the .NET Framework API documentation. Specifically, the general contract specifications for the Object.Equals() and Object.GetHashCode() methods are found in the documentation for the System.Object class.

Fundamental Behavior

Fundamental object behaviors include object creation, member accessibility, overriding the Object.ToString() method, and object serialization.

Object Creation – Constructors

Constructors are special methods that have the same name as the class in which they appear. Constructors have no return type, not even `void`. The purpose of a constructor is to properly initialize an object upon its instantiation.

Default Constructor

A default constructor has no method parameters. If you fail to define a default constructor, the C# compiler creates one for you. This ready-made default constructor may or may not be sufficient. The ready-made default constructor will most certainly not perform any custom field initializations. Any fields defined by your type will be initialized to their default values. Therefore, it's always a good idea to provide a default constructor, even if its body is empty. Implementing an empty-bodied default constructor says, "I know full well what I'm doing and I'm perfectly happy with the behavior provided by this empty default constructor."

Private Constructors

Most constructors you define will have public accessibility, but in some cases you'll want to declare constructors to be private. You'll find private constructors in singletons, where the class provides a static method that you use to request an object but disallows the creation of new objects willy-nilly by client software. I discuss the singleton and other software design patterns in *Chapter 25: Helpful Design Patterns*.

Overloaded Constructors

Constructor methods can be overloaded to allow the creation of objects in different ways. If your user-defined type is complex, you may find you need several overloaded constructors to accommodate the many possible ways to create objects.

Another good use for an overloaded constructor is to create what is referred to as a copy constructor. A copy constructor allows you to create an object of one type from an existing object of the same or different type. Copy constructors are covered later in the section titled Copy/Assignment Behavior.

Member Accessibility

The keywords `public`, `protected`, and `private` are used to control member accessibility. There are two avenues of member accessibility: *horizontal* and *vertical*.

Horizontal Member Access

Horizontal access is that access granted by an object of one type to an object of another type. Members declared to have public accessibility can be horizontally accessed. Members declared to be protected or private are not horizontally accessible.

© 2008 Rick Miller — All Rights Reserved

Vertical Member Access

Vertical access is that access granted by a base class object to derived class objects. Members declared to be public or protected are vertically accessible by a derived class. Private members are not vertically accessible.

Another way to think about vertical accessibility is to ask, "What gets inherited?" Members with public or protected accessibility get inherited but private members do not.

Figure 22-1 gives a diagram that illustrates the concepts of horizontal and vertical member accessibility.

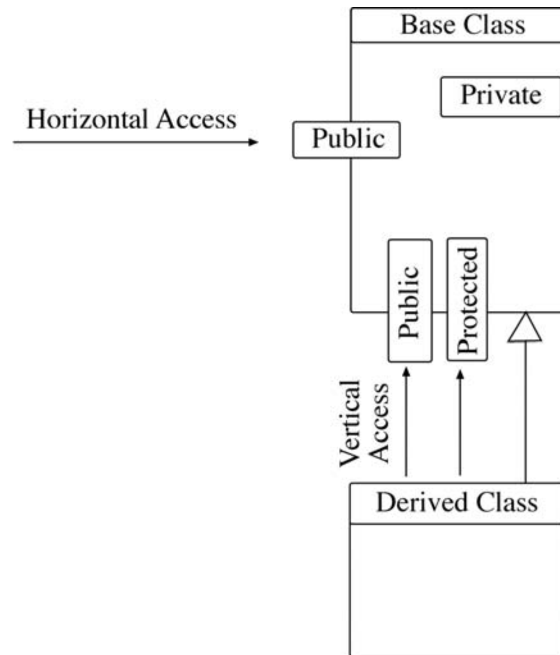


Figure 22-1: Horizontal and Vertical Member Accessibility

Referring to Figure 22-1 — public members are both horizontally and vertically accessible. Protected members are vertically accessible. Private members are neither horizontally nor vertically accessible.

Overriding Object.ToString()

The default implementation of the Object.ToString() method for user-defined types is to simply return the name of the type. In most situations this is rarely an adequate behavior. I recommend that you override the Object.ToString() method and make it do something useful. In most cases it's obvious what information about the object should be conveyed via the ToString() method. In those situations where it's not so obvious then a summary of the state of the object's fields is usually a good way to go. Also, as you'll see later, the overridden Object.ToString() method comes in handy when overriding the Object.Equals() and Object.GetHashCode() methods.

Static vs. Instance Members

The keyword *static* is used to declare class-wide members. For example, a static field is shared by all of a class's objects. A change to the static field's value by one object affects the value of the field in all objects.

By contrast, objects have their very own copies of non-static or instance fields. A change to the value of an instance field by one object does not affect the value of that field in another object because each object has its own personal copy of that field.

Static methods can only access static fields. Instance methods can access both static and instance fields.

Class-wide constants are declared with the *const* keyword. A const field is akin to a static readonly field but there is a difference as I explained in Chapter 9.

© 2008 Rick Miller — All Rights Reserved

Serialization

If you intend to send your user-defined type objects across the network, save them to a file, or convert them into some other form, say, XML, then you'll need to make them serializable. You do this by placing the Serializable attribute above the class declaration. The Serializable attribute enables the .NET runtime environment to automatically serialize an object, which, in most cases, works just file. The important thing to remember with serialization is that if you are serializing graphs of complex objects, then all the sub-objects must themselves be tagged with the Serializable attribute.

Prior to the .NET Framework version 2.0, if you wanted to customize the serialization process, you needed to implement the ISerializable interface, but this form of custom serialization has been superseded with the addition of several more attributes: *NonSerialized*, *OptionalField*, *OnSerializing*, *OnSerialized*, *OnDeserializing*, and *OnDeserialized*. The first two attributes, NonSerialized and OptionalField are applied to fields; the remaining four are applied to methods you want to execute during those four stages of object serialization. Let's look at an example.

Custom Serialization Example

I will use the PersonVO class from Chapter 20 to demonstrate custom serialization. The PersonVO class is listed in Example 22.1.

22.1 PersonVO.cs

```
1    using System;
2
3      [Serializable]
4      public class PersonVO {
5
6        //enumeration
7        public enum Sex {MALE, FEMALE};
8
9        // private instance fields
10       private String   _firstName;
11       private String   _middleName;
12       private String   _lastName;
13       private Sex      _gender;
14       private DateTime _birthday;
15
16       //default constructor
17       public PersonVO(){}
18
19       public PersonVO(String firstName, String middleName, String lastName,
20                      Sex gender, DateTime birthday){
21          FirstName = firstName;
22          MiddleName = middleName;
23          LastName = lastName;
24          Gender = gender;
25          BirthDay = birthday;
26       }
27
28       // public properties
29       public String FirstName {
30         get { return _firstName; }
31         set { _firstName = value; }
32       }
33
34       public String MiddleName {
35         get { return _middleName; }
36         set { _middleName = value; }
37       }
38
39       public String LastName {
40         get { return _lastName; }
41         set { _lastName = value; }
42       }
43
44       public Sex Gender {
45         get { return _gender; }
46         set { _gender = value; }
47       }
48
49       public DateTime BirthDay {
50         get { return _birthday; }
51         set { _birthday = value; }
52       }
53
```


© 2008 Rick Miller — All Rights Reserved

```
54      public int Age {
55        get {
56          int years = DateTime.Now.Year - _birthday.Year;
57          int adjustment = 0;
58            if(DateTime.Now.Month < _birthday.Month){
59              adjustment = 1;
60            }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
61                    adjustment = 1;
62                  }
63          return years - adjustment;
64        }
65      }
66
67        public String FullName {
68          get { return FirstName + " " + MiddleName + " " + LastName; }
69        }
70
71        public String FullNameAndAge {
72          get { return FullName + " " + Age; }
73        }
74
75        public override String ToString(){
76          return FullName + " is a " + Gender + " who is " + Age + " years old.";
77        }
78     } // end PersonVO class
```

The next example provides a short program that inserts some PersonVO objects into a list and then serializes the list. The program gives you three run options: create, append, and read. When run with the create option, it creates a new list and saves it to a new file. The append option deserializes the list, adds a PersonVO object to the list, then serializes the list. The read option deserializes the list and prints its contents to the console.

22.2 MainApp.cs

```
1     using System;
2     using System.IO;
3     using System.Collections.Generic;
4     using System.Runtime.Serialization;
5     using System.Runtime.Serialization.Formatters.Binary;
6
7     public class MainApp {
8       public static void Main(String[] args){
9         FileStream fs = null;
10        BinaryFormatter bf = null;
11        List<PersonVO> people_list = null;
12        PersonVO p1 = null;
13
14        if(args.Length > 0){
15          switch(args[0]){
16             case "create":
17                  people_list = new List<PersonVO>();
18                  p1 = new PersonVO("Rick", "Warren", "Miller", PersonVO.Sex.MALE, new DateTime(1964,7,8));
19                  people_list.Add(p1);
20                  fs = new FileStream("people.dat", FileMode.Create);
21                  bf = new BinaryFormatter();
22                  bf.Serialize(fs, people_list);
23                  fs.Close();
24                  break;
25
26             case "append":
27                  fs = new FileStream("people.dat", FileMode.Open);
28                  bf = new BinaryFormatter();
29                  people_list = (List<PersonVO>) bf.Deserialize(fs);
30                  fs.Close();
31                  p1 = new PersonVO("Rick", "Warren", "Miller", PersonVO.Sex.MALE, new DateTime(1964,7,8));
32                  people_list.Add(p1);
33                  fs = new FileStream("people.dat", FileMode.Create);
34                  bf = new BinaryFormatter();
35                  bf.Serialize(fs, people_list);
36                  fs.Close();
37                 break;
38
39             case "read":
40                 fs = new FileStream("people.dat", FileMode.Open);
41                 bf = new BinaryFormatter();
42                 people_list = (List<PersonVO>) bf.Deserialize(fs);
43                 foreach(PersonVO p in people_list){
44                   Console.WriteLine(p);
45                 }
46                 fs.Close();
47                 break;
48
```


© 2008 Rick Miller — All Rights Reserved

```
49            default: break;
50
51          } // end switch
52        } // end if
53      } // end Main
54    } // end class definition
```

To compile this code I've placed both files in a folder named Serialization and compiled them both together using the following command:

```
csc *.cs
```

Figure 22-2 shows the results of running this program several times, first with the create option, then a few times with the append option, and lastly with the read option.

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Serialization>mainapp create

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Serialization>mainapp append

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Serialization>mainapp append

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Serialization>mainapp append

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Serialization>mainapp append

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Serialization>mainapp append

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Serialization>mainapp read
Rick Warren Miller is a MALE who is 43 years old.
Rick Warren Miller is a MALE who is 43 years old.
Rick Warren Miller is a MALE who is 43 years old.
Rick Warren Miller is a MALE who is 43 years old.
Rick Warren Miller is a MALE who is 43 years old.
Rick Warren Miller is a MALE who is 43 years old.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Serialization>_
```

Figure 22-2: Running Example 22.2 Several Times

Alright, now suppose you want to make a modification to the PersonVO class. The new version of the application needs to be able to read previously serialized PersonVO objects without throwing an exception. As part of the modifications, you'd like to add several new fields, some new properties, and some other tweaks. Example 22.3 gives the modified PersonVO class.

22.3 PersonVO.cs (modified)

```
1    using System;
2    using System.Runtime.Serialization;
3
4      [Serializable]
5      public class PersonVO {
6
7        //enumeration
8        public enum Sex {MALE, FEMALE};
9        public enum Haircolor {BLONDE, BROWN, BLACK};
10
11       // private instance fields
12       private String   _firstName;
13       private String   _middleName;
14       private String   _lastName;
15       private Sex      _gender;
16       private DateTime _birthday;
17       [OptionalField]
18       private Haircolor _haircolor;
19       [OptionalField]
20       private DateTime _dateSerialized;
21       [OptionalField]
22       private DateTime _dateDeserialized;
23
24       //default constructor
25       public PersonVO(){}
26
27       public PersonVO(String firstName, String middleName, String lastName,
28                      Sex gender, DateTime birthday, Haircolor haircolor){
29          FirstName = firstName;
30          MiddleName = middleName;
31          LastName = lastName;
32          Gender = gender;
33          BirthDay = birthday;
34          HairColor = haircolor;
35       }
36
37       // public properties
```

 © 2008 Rick Miller — All Rights Reserved

```
38      public String FirstName {
39        get { return _firstName; }
40        set { _firstName = value; }
41      }
42
43      public String MiddleName {
44        get { return _middleName; }
45        set { _middleName = value; }
46      }
47
48      public String LastName {
49        get { return _lastName; }
50        set { _lastName = value; }
51      }
52
53      public Sex Gender {
54        get { return _gender; }
55        set { _gender = value; }
56      }
57
58      public DateTime BirthDay {
59        get { return _birthday; }
60        set { _birthday = value; }
61      }
62
63     public int Age {
64       get {
65         int years = DateTime.Now.Year - _birthday.Year;
66         int adjustment = 0;
67           if(DateTime.Now.Month < _birthday.Month){
68             adjustment = 1;
69           }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
70                     adjustment = 1;
71                 }
72         return years - adjustment;
73       }
74     }
75
76      #region New Properties
77      public Haircolor HairColor {
78        get { return _haircolor; }
79        set { _haircolor = value; }
80      }
81
82      public DateTime DateSerialized {
83        get { return _dateSerialized; }
84      }
85
86      public DateTime DateDeserialized {
87        get { return _dateDeserialized; }
88      }
89      #endregion
90
91      public String FullName {
92        get { return FirstName + " " + MiddleName + " " + LastName; }
93      }
94
95      public String FullNameAndAge {
96        get { return FullName + " " + Age; }
97      }
98
99      public override String ToString(){
100       return FullName + " is a " + Gender + " who is " + Age + " years old with " + HairColor +
101              " hair.\r\n" +
102              "-->Date Serialized:" + DateSerialized + ", Date Deserialized " + DateDeserialized;
103     }
104
105     #region Custom Serialization Methods
106     [OnSerializing]
107     internal void OnSerializingMethod(StreamingContext context){
108        _dateSerialized = DateTime.Now;
109     }
110
111     [OnDeserialized]
112     internal void OnDeserialized(StreamingContext context){
113       _dateDeserialized = DateTime.Now;
114     }
115     #endregion
116
117   } // end PersonVO class
```


© 2008 Rick Miller — All Rights Reserved

Referring to Example 22.3 — First, on line 2, I've added the `using` System.Runtime.Serialization directive. I've added three new fields starting on line 17 and above each one I've placed the OptionalField attribute. I modified the constructor to take another parameter named haircolor. Starting on line 74, I added three new properties named HairColor, DateSerialized, and DateDeserialized. The last two properties are readonly. Finally, starting on line 103, I've added two methods that let me modify the DateSerialized and DateDeserialized properties during the serialization and deserialization process.

The modified MainApp class appears in Example 22.4.

22.4 MainApp.cs (Mod 1)

```
1    using System;
2    using System.IO;
3    using System.Collections.Generic;
4    using System.Runtime.Serialization;
5    using System.Runtime.Serialization.Formatters.Binary;
6
7    public class MainApp {
8      public static void Main(String[] args){
9        FileStream fs = null;
10       BinaryFormatter bf = null;
11       List<PersonVO> people_list = null;
12       PersonVO p1 = null;
13
14       if(args.Length > 0){
15         switch(args[0]){
16            case "create":
17                 people_list = new List<PersonVO>();
18                 p1 = new PersonVO("Rick", "Warren", "Miller", PersonVO.Sex.MALE, new DateTime(1964,7,8),
19                                  PersonVO.Haircolor.BROWN);
20                 people_list.Add(p1);
21                 fs = new FileStream("people.dat", FileMode.Create);
22                 bf = new BinaryFormatter();
23                 bf.Serialize(fs, people_list);
24                 fs.Close();
25                 break;
26
27            case "append":
28                 fs = new FileStream("people.dat", FileMode.Open);
29                 bf = new BinaryFormatter();
30                 people_list = (List<PersonVO>) bf.Deserialize(fs);
31                 fs.Close();
32                 p1 = new PersonVO("Rick", "Warren", "Miller", PersonVO.Sex.MALE, new DateTime(1964,7,8),
33                                  PersonVO.Haircolor.BROWN);
34                 people_list.Add(p1);
35                 fs = new FileStream("people.dat", FileMode.Create);
36                 bf = new BinaryFormatter();
37                 bf.Serialize(fs, people_list);
38                 fs.Close();
39                break;
40
41            case "read":
42                fs = new FileStream("people.dat", FileMode.Open);
43                bf = new BinaryFormatter();
44                people_list = (List<PersonVO>) bf.Deserialize(fs);
45                foreach(PersonVO p in people_list){
46                  Console.WriteLine(p);
47                }
48                fs.Close();
49                break;
50
51             default: break;
52
53         } // end switch
54       } // end if
55     } // end Main
56   } // end class definition
```

Referring to Example 22.4 — the only changes I made here were to lines 18 and 32 to add the PersonVO.Haricolor.BROWN argument to the PersonVO constructor call.

To run this second experiment, I created a new folder named SerializationV2 to which I copied the MainApp.cs file and saved the modified PersonVO class there as well. I also copied the people.dat file created with the prior version of PersonVO.

Figure 22-3 shows the results of running MainApp in the read mode with the modified version of PersonVO. Remember, at this point the people.dat file contains serialized objects from the previous version of PersonVO.

Referring to Figure 22-3 — notice how the old PersonVO objects were deserialized without complaint when cast to the new version of PersonVO. Since the old PersonVO objects were not serialized with the _haircolor or

© 2008 Rick Miller — All Rights Reserved

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\SerializationV2>mainapp read
Rick Warren Miller is a MALE who is 43 years old with BLONDE hair.
-->Date Serialized:1/1/0001 12:00:00 AM, Date Deserialized 6/7/2008 4:07:15 PM
Rick Warren Miller is a MALE who is 43 years old with BLONDE hair.
-->Date Serialized:1/1/0001 12:00:00 AM, Date Deserialized 6/7/2008 4:07:15 PM
Rick Warren Miller is a MALE who is 43 years old with BLONDE hair.
-->Date Serialized:1/1/0001 12:00:00 AM, Date Deserialized 6/7/2008 4:07:15 PM
Rick Warren Miller is a MALE who is 43 years old with BLONDE hair.
-->Date Serialized:1/1/0001 12:00:00 AM, Date Deserialized 6/7/2008 4:07:15 PM
Rick Warren Miller is a MALE who is 43 years old with BLONDE hair.
-->Date Serialized:1/1/0001 12:00:00 AM, Date Deserialized 6/7/2008 4:07:15 PM

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\SerializationV2>
```

Figure 22-3: Running MainApp in the Read Mode

_dateSerialized fields, when the old objects are deserialized, these new fields are initialized to default values. In the case of the Haircolor enumeration, the default value is zero, which equates to the first enum value BLONDE. The _dateSerialized field is initialized to 1/1/0001.

Figure 22-4 shows the results of running MainApp in the append mode several times and then finally in the read mode. Compare this output against that of Figure 22-3 above.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\SerializationV2>mainapp read
Rick Warren Miller is a MALE who is 43 years old with BLONDE hair.
-->Date Serialized:6/7/2008 4:08:34 PM, Date Deserialized 6/7/2008 4:08:37 PM
Rick Warren Miller is a MALE who is 43 years old with BLONDE hair.
-->Date Serialized:6/7/2008 4:08:34 PM, Date Deserialized 6/7/2008 4:08:37 PM
Rick Warren Miller is a MALE who is 43 years old with BLONDE hair.
-->Date Serialized:6/7/2008 4:08:34 PM, Date Deserialized 6/7/2008 4:08:37 PM
Rick Warren Miller is a MALE who is 43 years old with BLONDE hair.
-->Date Serialized:6/7/2008 4:08:34 PM, Date Deserialized 6/7/2008 4:08:37 PM
Rick Warren Miller is a MALE who is 43 years old with BLONDE hair.
-->Date Serialized:6/7/2008 4:08:34 PM, Date Deserialized 6/7/2008 4:08:37 PM
Rick Warren Miller is a MALE who is 43 years old with BROWN hair.
-->Date Serialized:6/7/2008 4:08:34 PM, Date Deserialized 6/7/2008 4:08:37 PM
Rick Warren Miller is a MALE who is 43 years old with BROWN hair.
-->Date Serialized:6/7/2008 4:08:34 PM, Date Deserialized 6/7/2008 4:08:37 PM
Rick Warren Miller is a MALE who is 43 years old with BROWN hair.
-->Date Serialized:6/7/2008 4:08:34 PM, Date Deserialized 6/7/2008 4:08:37 PM
Rick Warren Miller is a MALE who is 43 years old with BROWN hair.
-->Date Serialized:6/7/2008 4:08:34 PM, Date Deserialized 6/7/2008 4:08:37 PM

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\SerializationV2>_
```

Figure 22-4: Results of Running MainApp Several More Times in the Append Mode then Read Mode

Quick Review

Things to think about regarding fundamental object behaviors include object creation, member accessibility, overriding the Object.ToString() method, the use of static vs. instance fields and methods, and custom serialization. It's a good idea to always provide a default constructor. There are two avenues of member accessibility to consider: *horizontal* and *vertical*. Control both with the keywords `public`, `protected`, and `private`. Override the Object.ToString() method to provide a customized string representation of the state of your object. Use the attributes *NonSerialized*, *OptionalField*, *OnSerializing*, *OnSerialized*, *OnDeserializing*, and *OnDeserialized* when you need to implement custom object serialization.

Copy/Assignment Behavior

In many programming situations you'll confront the need to make copies of objects. Copying comes in many forms. In some instances, you'll want to create a new object using data contained in an existing object. You can do this explicitly with the help of a *copy constructor*. However, in many cases, copying takes place implicitly behind the scenes, like when you pass objects as arguments to methods. The argument values are automatically copied to the parameters. (I discuss parameter passing in detail in Chapter 9.)

© 2008 Rick Miller — All Rights Reserved

If you want your objects to behave well in copy/assignment situations, you'll need to be aware of several important issues. These include the difference between copying value type vs. reference type objects, and the difference between a *shallow copy* vs. a *deep copy*. I will also show you how to create copy constructors.

Value Object vs. Reference Object Assignment

Remember! There's a difference between a value type variable and a reference type variable. A value type variable holds the actual data of the object whereas a reference type variable holds the address of the object. Also, you must use the `new` operator to create reference type objects. By now, you should know this by heart.

When you assign one value type object to another value type object (of the same type) the value represented by the right hand side object is copied to the left hand side object. For example, take a look at the following code snippet:

```
int i = 0;
int j = 1;
i = j;
```

In this case, the value of j is assigned to i. There are two distinct integer objects in the end: the one contained in i and the other contained in j.

A reference type variable contains an object's address. When you assign one reference to another you are copying addresses. For example, given the following code snippet:

```
Object o1 = new Object();
Object o2 = new Object();
o1 = o2;
```

In this case, reference o1 ultimately refers to the same object referenced by o2.

Rule Of Thumb – Favor The Class Construct For Complex Types

A good rule of thumb to apply when deciding whether to create a class or a structure is to let complexity be your guide. If your user-defined type is simple and contains only value types then a structure may be appropriate. That's because, as stated above, when assigning one value type object to another, everything from one gets copied to the other.

If, on the other hand, your user-defined type is complex or contains other reference types, favor the class construct. You incur less overhead when making assignments because you're only copying an address.

Shallow Copy vs. Deep Copy

Before you go copying objects, 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 one 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 original object's reference fields. Figure 22-5 illustrates the concept of the 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 a copy constructor or Clone() method.

A deep copy is different from a shallow copy in that it creates copies of the original object's instance field objects and assigns them to the new object. Figure 22-6 illustrates the concept of a deep copy. Referring to Figure 22-6 — before Object A is copied, its Object_Reference field points to Object B. A deep copy of Object A results 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.

You can easily see from these diagrams that an unintentional shallow copy can lead to unexpected object behavior

 © 2008 Rick Miller — All Rights Reserved

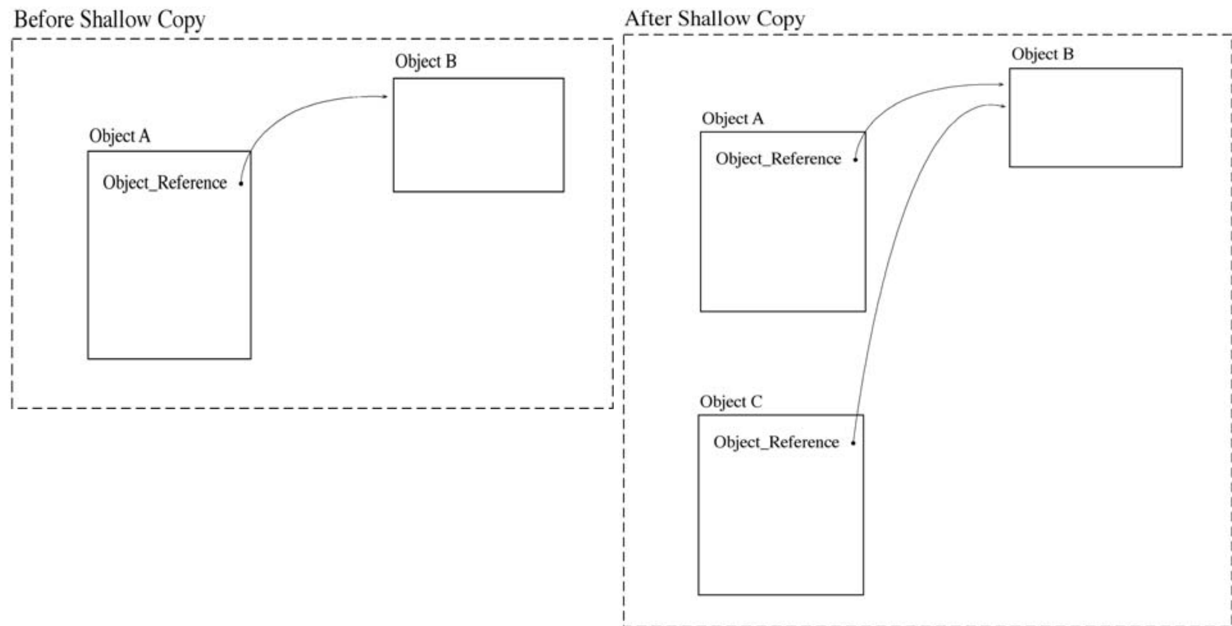


Figure 22-5: Concept of a Shallow Copy

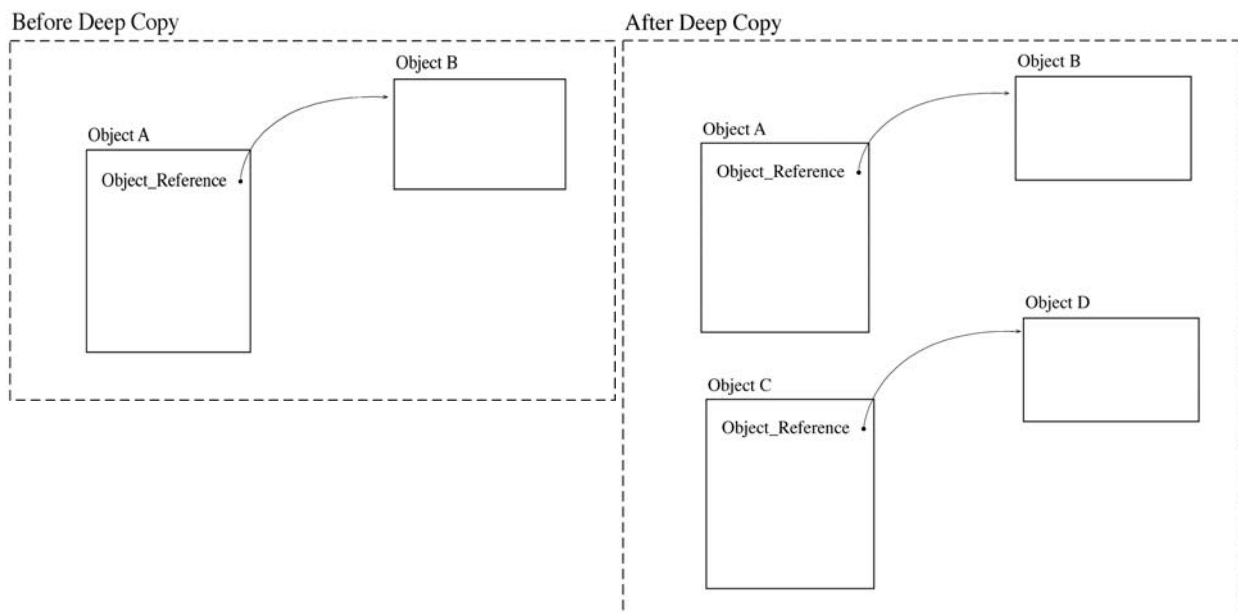


Figure 22-6: Concept of a Deep Copy

Copy Constructors

A copy constructor is used to create an object from another existing object of the same or different type. Example 22.5 lists the PersonVO class with the addition of a copy constructor that allows the creation of PersonVO objects from an existing PersonVO object.

© 2008 Rick Miller — All Rights Reserved

22.5 PersonVO.cs (with copy constructor)

```
1    using System;
2
3      [Serializable]
4      public class PersonVO {
5
6        //enumeration
7        public enum Sex {MALE, FEMALE};
8
9        // private instance fields
10       private String   _firstName;
11       private String   _middleName;
12       private String   _lastName;
13       private Sex      _gender;
14       private DateTime _birthday;
15
16       //default constructor
17       public PersonVO(){}
18
19       public PersonVO(String firstName, String middleName, String lastName,
20                       Sex gender, DateTime birthday){
21          FirstName = firstName;
22          MiddleName = middleName;
23          LastName = lastName;
24          Gender = gender;
25          BirthDay = birthday;
26       }
27
28       // copy constructor
29       public PersonVO(PersonVO person){
30         FirstName = person.FirstName;
31         MiddleName = person.MiddleName;
32         LastName = person.LastName;
33         Gender = person.Gender;
34         BirthDay = person.BirthDay;
35       }
36
37       // public properties
38       public String FirstName {
39         get { return _firstName; }
40         set { _firstName = value; }
41       }
42
43       public String MiddleName {
44         get { return _middleName; }
45         set { _middleName = value; }
46       }
47
48       public String LastName {
49         get { return _lastName; }
50         set { _lastName = value; }
51       }
52
53       public Sex Gender {
54         get { return _gender; }
55         set { _gender = value; }
56       }
57
58       public DateTime BirthDay {
59         get { return _birthday; }
60         set { _birthday = value; }
61       }
62
63      public int Age {
64       get {
65          int years = DateTime.Now.Year - _birthday.Year;
66          int adjustment = 0;
67            if(DateTime.Now.Month < _birthday.Month){
68              adjustment = 1;
69            }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
70                       adjustment = 1;
71                  }
72          return years - adjustment;
73        }
74      }
75
76       public String FullName {
77         get { return FirstName + " " + MiddleName + " " + LastName; }
78       }
79
```

 © 2008 Rick Miller — All Rights Reserved

```
80        public String FullNameAndAge {
81          get { return FullName + " " + Age; }
82        }
83
84        public override String ToString(){
85          return FullName + " is a " + Gender + " who is " + Age + " years old.";
86        }
87      } // end PersonVO class
```

Referring to Example 22.5 — the copy constructor begins on line 29 and takes a PersonVO object as an argument. The values are copied from the parameter to the current object's properties. Example 22.6 shows the copy constructor in use.

22.6 MainApp.cs

```
1     using System;
2
3     public class MainApp {
4       public static void Main(){
5         PersonVO p1 = new PersonVO("Rick", "Warren", "Miller", PersonVO.Sex.MALE, new DateTime(1968, 3, 7));
6         PersonVO p2 = new PersonVO(p1); // using copy constructor
7         Console.WriteLine(p1);
8         Console.WriteLine(p2);
9       }
10    }
```

Referring to Example 22.6 — a new PersonVO object is created on line 5 and its address is assigned to the reference p1. The reference p1 is then used on line 6 as an argument to the PersonVO copy constructor to create another PersonVO object. The results of running this program are shown in Figure 22-7.

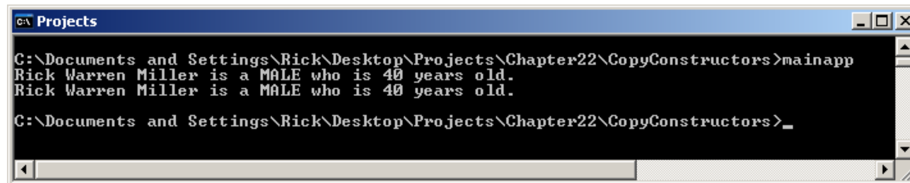


Figure 22-7: Results of Running Example 22.6

System.ICloneable vs. Object.MemberwiseClone()

Although the PersonVO class can be considered a complex type, in reality, it only contains simple fields consisting of String, DateTime, and the Sex enumeration, which is really an integer in the end. In this section I want to show you how to perform a deep copy of a complex object as well as highlight the differences between a deep copy vs. a shallow copy.

The .NET Framework provides two ways to clone an object. For value types or simple classes the Object.MemberwiseClone() method can be used straight out of the box to perform a shallow copy. If you need to exert more control over the cloning process you must implement the System.ICloneable interface.

The following example code shows a class named MyComplexType that has as a field an array of Strings. I implement the ICloneable interface to ensure that a proper deep copy is performed when calling the Clone() method.

22.7 MyComplexType.cs (implementing ICloneable)

```
1     using System;
2     using System.Text;
3     using System.Collections.Generic;
4
5     public class MyComplexType : ICloneable {
6       List<String> _string_list = null;
7       String _name = String.Empty;
8
9       public MyComplexType(String name){
10         Name = name;
11        _string_list = new List<String>();
12        for(int i = 0; i<5; i++){
13         _string_list.Add(String.Empty);
14       }
15      }
16
17      public List<String> StringList {
18          get { return _string_list; }
19      }
```


© 2008 Rick Miller — All Rights Reserved

```
20
21      public String Name {
22        get { return _name; }
23        set { _name = value; }
24      }
25
26      public Object Clone(){
27        MyComplexType mct = new MyComplexType(this.Name + " Clone");
28       for(int i = 0; i<this.StringList.Count; i++){
29         mct.StringList[i] = this.StringList[i];
30       }
31       return mct;
32      }
33
34      public Object GetMemberwiseClone(){
35        return this.MemberwiseClone();
36      }
37
38      public override String ToString(){
39        StringBuilder sb = new StringBuilder();
40        sb.Append(Name + ": ");
41        foreach(String s in StringList){
42          sb.Append(s + " ");
43        }
44        return sb.ToString();
45      }
46    } // end class definition
```

Referring to Example 22.7 — the Clone() method, as specified by the ICloneable interface, begins on line 26. The first thing I do is create a new instance of MyComplexType and give it the same name as the existing object. I add the string “clone” to the object’s name for demonstration purposes. Next, I iterate through the current object’s StringList and assign its values to the new object’s StringList. Finally, I return the new object.

On line 34, I created a method called GetMemberwiseClone() that uses the Object.MemberwiseClone() method to create a shallow copy of the current object.

Example 22.8 compares the operation of the Clone() and the GetMemberwiseClone() methods.

22.8 MainApp.cs

```
1     using System;
2
3     public class MainApp {
4       public static void Main(){
5         MyComplexType mct1 = new MyComplexType("mct1");
6         mct1.StringList[0] = "Hello";
7         mct1.StringList[1] = "World";
8         Console.WriteLine(mct1);
9
10        MyComplexType mct2 = (MyComplexType)mct1.Clone();
11        Console.WriteLine(mct2);
12        mct2.StringList[0] = "New String";
13        Console.WriteLine(mct1);
14        Console.WriteLine(mct2);
15
16        MyComplexType mct3 = (MyComplexType)mct2.GetMemberwiseClone();
17        Console.WriteLine(mct3);
18        mct2.StringList[2] = "Another String";
19        Console.WriteLine(mct2);
20        Console.WriteLine(mct3);
21      } // end main
22    } // end class definition
```

Referring to Example 22.8 — on line 5, I create an instance of MyComplexType named mct1. I then set the first two elements in its StringList to “Hello” and “World” respectively. I then write the string representation of mct1 to the console. (See MyComplexType’s ToString() method.)

On line 10, I declare a new reference named mct2 and assign to it a cloned instance of mct1 via the Clone() method. I then write mct2 to the console, which will look like mct1 but with the string “Clone” attached to its name.

Next, to demonstrate that both objects have their very own copy of the _string_list field, I set the first element of mct2’s StringList to the value “New String”. I then print out both mct1 and mct2 again to show they contain different values.

Finally, I declare one last reference named mct3 and use the GetMemberwiseClone() method to create a shallow copy of mct2. I them add another string to mct2’s StringList and then print both mct2 and mct3 to show they both share a common _string_list field.

 © 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\ICloneable>mainapp
mct1: Hello World
mct1 Clone: Hello World
mct1: Hello World
mct1 Clone: New String World
mct1 Clone: New String World
mct1 Clone: New String World Another String
mct1 Clone: New String World Another String

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\ICloneable>_
```

Figure 22-8: Results of Running Example 22.8

Quick Review

When implementing copying behavior, you must be aware of the difference between a *shallow copy* and a *deep copy*. There are generally three ways to implement copying or cloning behavior: create a copy constructor, implement the ICloneable interface, or use the Object.MemberwiseClone() method. Copy constructors and the ICloneable interface let you control the object copying process and, if necessary, allows you to implement a deep copy. The Object.MemberwiseClone() method performs a shallow copy only and is best utilized on simple value types.

Equality Behavior

If you need to compare user-defined type objects against each other for equality you'll need to be aware of equality behavior. In *Chapter 21: Operator Overloading*, you learned that if you overload the == and != operators you should also override the Object.Equals() and Object.GetHashCode() methods. In this section, I focus on the Object.Equals() and Object.GetHashCode() methods, show you how to override them with the help of the Object.ToString() method, and present you with a list of rules you should follow when overriding these methods.

Reference Equality vs. Value Equality

Normally, when you compare two reference objects for equality like this...

```
o1 == o2
```

...you are comparing their addresses. In other words, if o1 and o2 refer the to the same location in memory then they must be equal because they refer to the same object.

Comparing value objects for equality is different in that a test for equality compares their values. For example, given two integer variables:

```
int i = 1;
int j = 2;
```

The expression (`i == j`) compares the value of i, which is 1, against the value of j, which is 2. In either case I can substitute the == operator with the Equals() method like so:

```
o1.Equals(o2)
i.Equals(j);
```

The results would be the same. However, in the case of reference objects, it's not always desirable behavior to strictly use an object's address as a basis for equality. Take strings for example. Two strings of equal value may be different objects as the following code snippet suggests:

```
String s1 = "Hello";
String s2 = "Hello";
```

So the expression (`s1 == s2`) will yield true just as `s1.Equals(s2)` will yield true. This is because the == operator has been overloaded to perform a value or string content comparison, which is what you'd expect when

© 2008 Rick Miller — All Rights Reserved

comparing two strings. (**Note:** This is different from the way the **==** operator behaves in Java, which is why you'll see Java programmers comparing strings for equality using the String.Equals() method in C#.)

Rules For Overriding The Object.Equals() Method

When overriding the Object.Equals() method, you must ensure that it subscribes to the expected behavior as specified in the .NET Framework documentation. Table 22.2 lists the required behaviors of an overridden Object.Equals() method. (**Note:** Your overloaded **==** operator should work the same way! See *Chapter 21: Operator Overloading*.)

Should be...	Rule	Comment
Reflexive	x.Equals(x) returns true	Exception: floating-point types
Symmetric	x.Equals(y) returns the same as y.Equals(x)	
Transitive	(x.Equals(y) && y.Equals(z)) returns true if and only if x.Equals(z) returns true	
Consistent	Successive calls to x.Equals(y) return the same value as long as the objects referenced by x and y remain unchanged.	
	x.Equals(null) returns false	Or a null reference
	x.Equals(y) returns true if both x and y are NaN	NaN means Non a Number
	Calls to Object.Equals() must not throw exceptions.	No exceptions!
	Override the Object.GetHashCode() method.	If you override the Object.Equals() method.

Table 22-2: Rules for Overriding Object.Equals() method

Overriding The Object.GetHashCode() Method

When you override the Object.Equals() method, you should also override the Object.GetHashCode() method to ensure proper object behavior. This section presents two approaches to implementing a suitable GetHashCode() method. Now, don't be alarmed when I reference two very good Java books. The techniques used to create a suitable hashcode algorithm apply equally to C# as well as Java.

The GetHashCode() method returns an integer which is referred to as the object's hash value. The default implementation of GetHashCode() 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 is 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 GetHashCode() method and obey the general contract as specified in the .NET Framework API documentation. The general contract for the GetHashCode() is given in Table 22-3.

Check	Criterion
	The GetHashCode() method must consistently return the same integer when invoked on the same object more than once during an execution of a C# or .NET 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.

Table 22-3: The GetHashCode() General Contract

© 2008 Rick Miller — All Rights Reserved

Check	Criterion
	The GetHashCode() method must produce the same results when called on two objects if they are equal according to the Equals() method.
	The GetHashCode() method is not required to return distinct integer results for logically unequal objects, however, failure to do so may result in degraded hash table performance.

Table 22-3: The GetHashCode() General Contract

As you can see from Table 22-3, there is a close relationship between the Object.Equals() and Object.GetHashCode() 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 GetHashCode() method is to have it return the same value consistently for logically equal objects. It would also be nice if the GetHashCode() 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.GetHashCode() method, I want to provide you with two hash code generation algorithms. These algorithms come from two excellent Java references. I have changed the text to reflect the .NET method names Object.Equals() and Object.GetHashCode() respectively, and have converted Java operations into compatible C# .NET operations.

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 (bool) 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: `(unsigned)(f^(f >> 32))`
      - iv. If the field is a float compute: `Convert.ToInt32(f)`
      - v. If the field is a double compute: `Convert.ToInt64(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 GetHashCode() on the field. If a more complex comparison is required, compute a "canonical representation" for this field and invoke GetHashCode() 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 GetHashCode() method on that string.
3. Return the resulting hash code value.

© 2008 Rick Miller — All Rights Reserved

Overridng Object.Equals() and Object.GetHashCode() Methods In The PersonVO Class

Example 22.9 shows how to apply Ashmore's hash code algorithm to PersonVO objects.

22.9 PersonVO.cs (overriden Equals() and GetHashCode())

```
1    using System;
2
3      [Serializable]
4      public class PersonVO {
5
6        //enumeration
7        public enum Sex {MALE, FEMALE};
8
9        // private instance fields
10       private String   _firstName;
11       private String   _middleName;
12       private String   _lastName;
13       private Sex      _gender;
14       private DateTime _birthday;
15
16       //default constructor
17       public PersonVO(){}
18
19       public PersonVO(String firstName, String middleName, String lastName,
20                    Sex gender, DateTime birthday){
21          FirstName = firstName;
22          MiddleName = middleName;
23          LastName = lastName;
24          Gender = gender;
25          BirthDay = birthday;
26       }
27
28       // copy constructor
29       public PersonVO(PersonVO person){
30         FirstName = person.FirstName;
31         MiddleName = person.MiddleName;
32         LastName = person.LastName;
33         Gender = person.Gender;
34         BirthDay = person.BirthDay;
35       }
36
37       // public properties
38       public String FirstName {
39         get { return _firstName; }
40         set { _firstName = value; }
41       }
42
43       public String MiddleName {
44         get { return _middleName; }
45         set { _middleName = value; }
46       }
47
48       public String LastName {
49         get { return _lastName; }
50         set { _lastName = value; }
51       }
52
53       public Sex Gender {
54         get { return _gender; }
55         set { _gender = value; }
56       }
57
58       public DateTime BirthDay {
59         get { return _birthday; }
60         set { _birthday = value; }
61       }
62
63      public int Age {
64       get {
65          int years = DateTime.Now.Year - _birthday.Year;
66          int adjustment = 0;
67            if(DateTime.Now.Month < _birthday.Month){
68              adjustment = 1;
69            }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
70                      adjustment = 1;
71                   }
72          return years - adjustment;
73        }
74      }
```

 © 2008 Rick Miller — All Rights Reserved

```
75
76       public String FullName {
77         get { return FirstName + " " + MiddleName + " " + LastName; }
78       }
79
80       public String FullNameAndAge {
81         get { return FullName + " " + Age; }
82       }
83
84       public override String ToString(){
85         return FullName + " is a " + Gender + " who is " + Age + " years old.";
86       }
87
88       // override System.Object methods
89       public override bool Equals(Object o){
90         if(o == null){
91           return false;
92         }
93         return this.ToString().Equals(o.ToString());
94       }
95
96       public override int GetHashCode(){
97         return this.ToString().GetHashCode();
98       }
99
100    } // end PersonVO class
```

Referring to Example 22.9 — The overridden Object.Equals() method starts on line 87. If the parameter is null, I return false. If this test wasn't in place, the code would throw an exception and, according to the contract, that's not good. To make the actual comparison, I simply compare their string representations and return the result.

The GetHashCode() method starts on line 94. It simply calls GetHashCode() on the object's ToString() value and returns the result.

Example 22.10 shows these two methods in action.

22.10 MainApp.cs

```
1     using System;
2
3     public class MainApp {
4       public static void Main(){
5         PersonVO p1 = new PersonVO("Rick", "Warren", "Miller", PersonVO.Sex.MALE, new DateTime(1964, 3, 7));
6         PersonVO p2 = new PersonVO(p1);
7         PersonVO p3 = new PersonVO("Coralie", "Sarah", "Miller", PersonVO.Sex.FEMALE,
8                       new DateTime(1968, 4, 5));
9         PersonVO p4 = null;
10
11        Console.WriteLine("p1.Equals(p1) = " +  p1.Equals(p1));
12        Console.WriteLine("p1.Equals(p2) = " +  p1.Equals(p2));
13        Console.WriteLine("p2.Equals(p1) = " +  p2.Equals(p1));
14        Console.WriteLine("p1.Equals(p3) = " +  p1.Equals(p3));
15        Console.WriteLine("p3.Equals(p1) = " +  p3.Equals(p1));
16        Console.WriteLine("p1.Equals(null) = " +  p1.Equals(null));
17        Console.WriteLine("p1.Equals(p4) = " +  p1.Equals(p4));
18        Console.WriteLine("------------------------------------");
19        Console.WriteLine("p1.GetHashCode() = " + p1.GetHashCode());
20        Console.WriteLine("p2.GetHashCode() = " + p2.GetHashCode());
21        Console.WriteLine("p3.GetHashCode() = " + p3.GetHashCode());
22      }
23    }
```

Figure 22-9 shows the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\EqualsHashcodeMethods>mainapp
p1.Equals(p1) = True
p1.Equals(p2) = True
p2.Equals(p1) = True
p1.Equals(p3) = False
p3.Equals(p1) = False
p1.Equals(null) = False
p1.Equals(p4) = False
------------------------------------
p1.GetHashCode() = -847669783
p2.GetHashCode() = -847669783
p3.GetHashCode() = -1136128382

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\EqualsHashcodeMethods>_
```

Figure 22-9: Results of Running Example 22.10

© 2008 Rick Miller — All Rights Reserved

Quick Review

Keep in mind, when implementing equality behavior, the differences between value and reference types. An overloaded Object.Equals() method should be *reflexive*, *symmetric*, *transitive*, and *consistent*. Also, it should not throw an exception when called.

Always override the Object.GetHashCode() method when you override Object.Equals(). Adhere to the general contract when implementing a hash code algorithm.

Comparison/Ordering Behavior

You'll often need to compare objects against each other to produce some type of ordering. If objects have a natural ordering then implement the IComparable<T> interface. You can do this if you own the source code. If your objects can be ordered in different ways at different times or you don't own the source code, extend the Comparer<T> class to create one or more custom comparer objects This section repeats material originally presented in *Chapter 14: Collections*. So, before you write me a nasty email, I want to let you know that I'm using the Person class vice the PersonVO class I've used earlier in this chapter.

Implementing System.IComparable<T>

Fundamental data types provided by the .NET Framework implement the System.IComparable or System.IComparable<T> interface and are therefore sortable. When you define a user-defined class or structure, you must ask yourself, as part of the design process, "Will objects of this type be compared with each other or with other types of objects?" If yes, then that class or structure should implement the IComparable<T> interface.

The IComparable<T> interface declares one method, CompareTo(). This method is automatically called during collection or array sort operations. If you want your user-defined types to sort correctly, you must provide an implementation for the CompareTo() method. How one object is compared against another is entirely up to you since you're the programmer.

Example 22.11 shows the Person class modified to implement the IComparable<T> interface.

22.11 Person.cs (implementing IComparable<T>)

```
1    using System;
2
3    public class Person : IComparable<Person> {
4
5      //enumeration
6      public enum Sex {MALE, FEMALE};
7
8      // private instance fields
9      private String   _firstName;
10     private String   _middleName;
11     private String   _lastName;
12     private Sex      _gender;
13     private DateTime _birthday;
14
15
16     //private default constructor
17     private Person(){}
18
19     public Person(String firstName, String middleName, String lastName,
20                   Sex gender, DateTime birthday){
21        FirstName = firstName;
22        MiddleName = middleName;
23        LastName = lastName;
24        Gender = gender;
25        BirthDay = birthday;
26     }
27
28     // public properties
29     public String FirstName {
30       get { return _firstName; }
31       set { _firstName = value; }
32     }
33
34     public String MiddleName {
```

 © 2008 Rick Miller — All Rights Reserved

```
35        get { return _middleName; }
36        set { _middleName = value; }
37      }
38
39      public String LastName {
40        get { return _lastName; }
41        set { _lastName = value; }
42      }
43
44      public Sex Gender {
45        get { return _gender; }
46        set { _gender = value; }
47      }
48
49      public DateTime BirthDay {
50        get { return _birthday; }
51        set { _birthday = value; }
52      }
53
54       public int Age {
55         get {
56           int years = DateTime.Now.Year - _birthday.Year;
57           int adjustment = 0;
58             if(DateTime.Now.Month < _birthday.Month){
59               adjustment = 1;
60             }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
61                     adjustment = 1;
62                   }
63           return years - adjustment;
64         }
65       }
66
67      public String FullName {
68        get { return FirstName + " " + MiddleName + " " + LastName; }
69      }
70
71      public String FullNameAndAge {
72        get { return FullName + " " + Age; }
73      }
74
75      public override String ToString(){
76        return FullName + " is a " + Gender + " who is " + Age + " years old.";
77      }
78
79      public int CompareTo(Person other){
80          return this.BirthDay.CompareTo(other.BirthDay);
81       }
82
83    } // end Person class
```

Referring to Example 22.11— the Person class implements the IComparable<T> interface. The CompareTo() method, starting on line 79, defines how one Person object is compared with another. In this case, I am comparing their BirthDay properties. Note that since the DateTime structure implements the IComparable<T> interface (*i.e.*, IComparable<DateTime>) one simply needs to call its version of the CompareTo() method to make the required comparison. But what's getting returned? Good question, and it's one that's answered in the next section.

Rules For Implementing The CompareTo(T other) Method

Table 22-4 lists the rules for implementing the CompareTo(T other) method.

Return Value	Returned When...
Less than Zero (-1)	This object is less than the *other* parameter
Zero (0)	This object is equal to the *other* parameter
Greater than Zero (1)	This object is greater than the *other* parameter, or, the *other* parameter is null

Table 22-4: Rules For Implementing IComparable<T>.CompareTo(T other) Method

© 2008 Rick Miller — All Rights Reserved

What property, exactly, you compare between objects is strictly dictated by the program design. In the case of the Person class I chose to compare birthdays. In the next section, I'll show you how you would compare last names.

Now that the Person class implements the IComparable<T> interface, Person objects can be compared against each other. Example 22.12 shows the List<T>.Sort() method in action.

22.12 SortingListDemo.cs

```
1    using System;
2    using System.Collections.Generic;
3
4    public class SortingListDemo {
5      public static void Main(){
6        List<Person> surrealists = new List<Person>();
7
8        Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
9        Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
10       Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
11       Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
12       Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
13       Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
14                              new DateTime(1887, 07, 28));
15
16       surrealists.Add(p1);
17       surrealists.Add(p2);
18       surrealists.Add(p3);
19       surrealists.Add(p4);
20       surrealists.Add(p5);
21       surrealists.Add(p6);
22
23       for(int i=0; i<surrealists.Count; i++){
24         Console.WriteLine(surrealists[i].FullNameAndAge);
25       }
26
27       surrealists.Sort();
28       Console.WriteLine("--------------------------------------");
29
30       for(int i=0; i<surrealists.Count; i++){
31         Console.WriteLine(surrealists[i].FullNameAndAge);
32       }
33
34     } // end Main()
35   } // end SortingListDemo
```

Referring to Example 22.12 — the Sort() method is called on line 27. Figure 22-10 shows the results of running this program.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Sorting_List>sortinglistdemo
Rick  Miller 47
Max  Ernst 117
Andre  Breton 112
Roland  Penrose 108
Lee  Miller 101
Henri-Robert-Marcel  Duchamp 121
---------------------------------------
Henri-Robert-Marcel  Duchamp 121
Max  Ernst 117
Andre  Breton 112
Roland  Penrose 108
Lee  Miller 101
Rick  Miller 47

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Sorting_List>
```

Figure 22-10: Results of Running Example 22.12

Extending The Comparer<T> Class

What if you don't own the source code to the objects you want to compare? No problem, simply implement a custom comparer object by extending the System.Collections.Generic.Comparer<T> class and provide an implementation for its Compare(T x, T y) method. Example 22.13 shows how a custom comparer might look. This particular example compares two Person objects.

22.13 PersonComparer.cs

```
1    using System.Collections.Generic;
2
3    public class PersonComparer : Comparer<Person> {
4
```


© 2008 Rick Miller — All Rights Reserved

```
5      /***********************************
6         Return -1 if p1 < p2 or p1 == null
7         Return  0 if p1 == p2
8         Return +1 if p1 > p2 or p2 == null
9      ***********************************/
10     public override int Compare(Person p1, Person p2){
11        if(p1 == null) return -1;
12        if(p2 == null) return  1;
13
14        return p1.LastName.CompareTo(p2.LastName);
15     }
16   }
```

Referring to Example 22.13— the PersonComparer class extends Comparer<T> (*i.e.*, Comparer<Person>) and provides an overriding implementation for its Compare(T x, T y) method. In this example I have renamed the parameters p1 and p2. Note that since I am comparing strings, I can simply call the String.CompareTo() method to actually perform the comparison.

Example 22.14 shows how the PersonComparer class is used to sort a list of Person objects.

22.14 ComparerSortDemo.cs

```
1    using System;
2    using System.Collections.Generic;
3
4    public class ComparerSortDemo {
5      public static void Main(){
6        List<Person> surrealists = new List<Person>();
7
8        Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
9        Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
10       Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
11       Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
12       Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
13       Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
14                                 new DateTime(1887, 07, 28));
15
16       surrealists.Add(p1);
17       surrealists.Add(p2);
18       surrealists.Add(p3);
19       surrealists.Add(p4);
20       surrealists.Add(p5);
21       surrealists.Add(p6);
22
23       for(int i=0; i<surrealists.Count; i++){
24         Console.WriteLine(surrealists[i].FullNameAndAge);
25       }
26
27       surrealists.Sort(new PersonComparer());
28       Console.WriteLine("---------------------------------------");
29
30       for(int i=0; i<surrealists.Count; i++){
31         Console.WriteLine(surrealists[i].FullNameAndAge);
32       }
33
34     } // end Main()
35   } // end SortingListDemo
```

Referring to Example 22.14 — an overloaded version of the List<T>.Sort() method is called on line 27. This version of the Sort() method takes a Comparer<T> object as an argument. Figure 22-11 shows the results of running this program.

```
Projects

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Sorting_with_Comparer>comparersortdemo
Rick  Miller 47
Max  Ernst 117
Andre  Breton 112
Roland  Penrose 108
Lee  Miller 101
Henri-Robert-Marcel  Duchamp 121
---------------------------------------
Andre  Breton 112
Henri-Robert-Marcel  Duchamp 121
Max  Ernst 117
Lee  Miller 101
Rick  Miller 47
Roland  Penrose 108

C:\Documents and Settings\Rick\Desktop\Projects\Chapter22\Sorting_with_Comparer>_
```

Figure 22-11: Results of Running Example 22.14

© 2008 Rick Miller — All Rights Reserved

Quick Review

If your user-defined type objects must be compared against each other for ordering (sorting), then implement either the IComparable<T> interface to create a natural ordering or extend the Comparer<T> class and create one or more comparer objects that will let you order your objects in different ways.

Summary

Things to think about regarding fundamental object behaviors include object creation, member accessibility, overriding the Object.ToString() method, the use of static vs. instance fields and methods, and custom serialization. It's a good idea to always provide a default constructor. There are two avenues of member accessibility to consider: *horizontal* and *vertical*. Control both with the keywords `public`, `protected`, and `private`. Override the Object.ToString() method to provide a customized string representation of the state of your object. Use the attributes *NonSerialized*, *OptionalField*, *OnSerializing*, *OnSerialized*, *OnDeserializing*, and *OnDeserialized* when you need to implement custom object serialization.

When implementing copying behavior, you must be aware of the difference between a *shallow copy* and a *deep copy*. There are generally three ways to implement copying or cloning behavior: create a copy constructor, implement the ICloneable interface, or use the Object.MemberwiseClone() method. Copy constructors and the ICloneable interface let you control the object copying process and, if necessary, allows you to implement a deep copy. The Object.MemberwiseClone() method performs a shallow copy only and is best utilized on simple value-types.

Keep in mind, when implementing equality behavior, the differences between value and reference types. An overloaded Object.Equals() method should be *reflexive*, *symmetric*, *transitive*, and *consistent*. Also, it should not throw an exception when called.

Always override the Object.GetHashCode() method when you override Object.Equals(). Adhere to the general contract when implementing a hash code algorithm.

If your user-defined type objects must be compared against each other for ordering (sorting), then implement either the IComparable<T> interface to create a natural ordering or extend the Comparer<T> class and create one or more comparer objects that will let you order your objects in different ways.

Skill-Building Exercises

1. **API Drill:** Research the System.Object class and read the rules for overriding the Object.Equals() and Object.GetHashCode() methods.

2. **Programming Drill:** Compile and run the programs listed in this chapter.

Suggested Projects

1. **EmployeeVO Serialization Do-Over:** How might the problem of the non-serializable image encountered in Chapter 20 have been solved with custom serialization?

© 2008 Rick Miller — All Rights Reserved

Self-Test Questions

1. What are the seven object usage scenarios?

2. What's the difference between a *shallow copy* and a *deep copy*?

3. Why do you think a shallow copy of a complex object may lead to unexpected object behavior?

4. (T/F) Static methods can access instance fields. Explain your answer.

5. (T/F) Private members allow vertical access. Explain your answer.

6. What rules, if any, should you apply when overriding the Object.Equals() method?

7. When overriding the Object.Equals() method, what other method must you override?

8. If you own the source code and your user-defined type objects have a natural ordering, what interface should you implement?

9. What can you do to order objects if you don't own the source code or you want to provide multiple orderings?

10. What's the difference between implementing ICloneable and simply using Object.MemberwiseClone() to clone objects?

References

Joshua Bloch. *Effective Java™ Programming Language Guide*. Addison-Wesley, Boston, MA. ISBN: 0-201-31005-8.

Microsoft Developer Network (MSDN) *.NET Framework 3.0 and3.5 Reference Documentation* [www.msdn.com]

Derek Ashmore. *The J2EE Architect's Handbook: How To Be A Successful Technical Architect For J2EE Applications. DVT Press, Lombard, IL. ISBN: 0972954899*

© 2008 Rick Miller — All Rights Reserved

Notes

© 2008 Rick Miller — All Rights Reserved

Chapter 23

Contax T / Kodax Tri-X

Nunnery – Chichen Itza , Mexico

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 (DbC) programming*
- *Describe the close relationship between the Liskov Substitution Principle and Design by Contract*
- *State the purpose of class invariants*
- *State the purpose of method preconditions and postconditions*
- *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 (OCP)*
- *State the purpose and use of the Dependency Inversion Principle (DIP)*

© 2008 Rick Miller — All Rights Reserved

Introduction

Building complex, well-behaved, object-oriented software is a difficult task for several reasons. First, simply programming in C# 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 C# programming language, will significantly improve you 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 (inheritance) 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.

 © 2008 Rick Miller — All Rights Reserved

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 names 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 subcontractor (subtype). Programmers should not need any a priori 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 LSB 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.

C# Support For The LSP And DbC

With the exception of type checking, C# 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.

© 2008 Rick Miller — All Rights Reserved

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 C#, class declaration and definition is usually combined. A class that specifies behavior only is known in C# 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. I discuss their definitions and application in this section.

Class Invariant

A class invariant is an assertion about an object property that must hold true for all valid states the 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.

© 2008 Rick Miller — All Rights Reserved

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 <= speed <= 800 holds true when the increment method completes its operations.

An Example

Example 23.1 gives the code for a class named Incrementer. An incrementer object can 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 with the help of the Debug.Assert() method. The Debug class is found in the System.Diagnostics namespace.

23.1 Incrementer.cs

```
1    #define DEBUG
2    using System;
3    using System.Diagnostics;
4
5    public class Incrementer {
6      /**********************************************
7       Class invariant: 0 <= Incrementer.val <= 100
8      *********************************************/
9      private int val = 0;
10
11     /*********************************************
12       Constructor Method: Incrementor(int i)
13       precondition: ((0 <= i) &&  (i <= 100))
14       postcondition:  0 <= Incrementer.val <= 100
15     ********************************************/
16     public Incrementer(int i){
17       Debug.Assert((0 <= i) && (i <= 100));
18       val = i;
19       Console.WriteLine("Incrementer object created with initial value of: " + val);
20       CheckInvariant();   // enforce class invariant
21     }
22
23     /**********************************************
24       Method: void Increment(int i)
25       precondition: 0 < i <= 5
26       postcondition: 0 <= Incrementer.val <= 100
27    *********************************************/
28     public virtual void Increment(int i){
29       Debug.Assert((0 < i) && (i <= 5)); // enforce precondition
30
31       if((val+i) <= 100){
32         val += i;
33       }else{
34         int temp = val;
35         temp += i;
36         val = (temp - 100);
37        }
38
39       CheckInvariant();           // enforce class invariant
40       Console.WriteLine("Incremeter value is: " + val);
41     }
42
43    /*********************************************
44      Method: void CheckInvariant() - called
45      immediately after any change to class
46      invariant to ensure invariant condition
47      is satisfied.
48     ******************************************/
49     private void CheckInvariant(){
50       Debug.Assert((0 <= val) && (val <= 100));
51     }
52   }// end Incrementer class definition
```

Referring to Example 24.1 — First, in order to use the Debug.Assert() method, you'll need to add the #define DEBUG directive at the top of the source file as I've done here.

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

© 2008 Rick Miller — All Rights Reserved

invariant is enforced with the Debug.Assert() method 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 preconditions 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 completes the class invariant state must be valid. The Increment() method's precondition is checked by the Debug.Assert() method on line 29. The class invariant is checked by calling the CheckInvariant() method on line 39.

Example 24.2 gives a short program putting the Incrementer class through its paces.

23.2 MainTestApp.cs

```
1    using System;
2
3    public class MainTestApp {
4      public static void Main(){
5        Incrementer i1 = new Incrementer(0);
6        i1.Increment(1);
7        i1.Increment(2);
8        i1.Increment(3);
9        i1.Increment(4);
10       i1.Increment(5);
11       i1.Increment(6); // throws an assesrtion exception
12    } // end Main() method
13   }// end MainTestApp clas definition
```

Referring to Example 23.2 — an Incrementer reference named i1 is declared and initialized on line 5. On lines 6 through 11, I call the Increment method via i1 with different increment values 1, 2, 3, 4, 5, and 6. If you put the #define DEBUG directive at the top of the Incrementer class's source file then the assertion will fail when line 11 executes, causing the Assertion Failed dialog window to display when the program executes. Figure 23-1 shows the results of running this program.

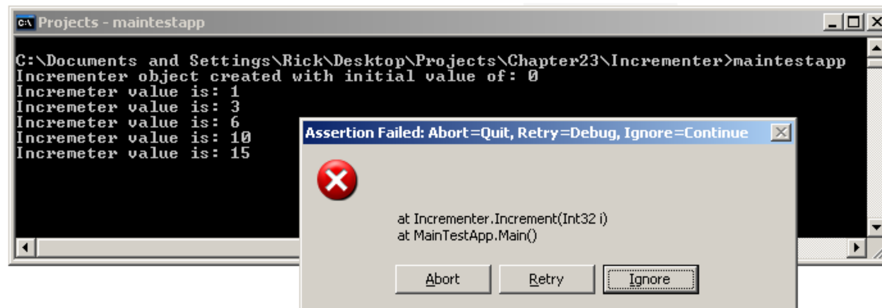


Figure 23-1: Results of Running Example 23.2

A Note On Using The Debug.Assert() Method To Enforce Pre- and Postconditions

As was just demonstrated, you can place the #define DEBUG directive at the top of your source files to enable the use of the System.Diagnostics.Debug.Assert() method. You can alternatively use the `/d:DEBUG` compiler switch when you compile your code.

The use of the assertion mechanism to enforce method preconditions 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 23.2. 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

 © 2008 Rick Miller — All Rights Reserved

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 23.3 gives the code for a class named DerivedIncrementer that extends the functionality of the Incrementer class.

23.3 DerivedIncrementer.cs

```
1    #define DEBUG
2    using System;
3    using System.Diagnostics;
4
5    public class DerivedIncrementer : Incrementer {
6      /*********************************************
7        Class invariant: 0 <= val <= 50
8      ********************************************/
9      private int val = 0;
10
11     /***********************************************
12      Constructor Method: DerivedIncrementer(int i)
13      precondition: ((0 <= i) && (i <= 50))
14      postcondition: 0 <= val <= 50
15     **********************************************/
16     public DerivedIncrementer(int i):base(i){
17       Debug.Assert((0 <= i) && (i <= 50));  // enforce precondition
18       val = i;
19       Console.WriteLine("DerivedIncrementer object created with value: " + val);
20       CheckInvariant();
21     }
22
23     /**********************************************
24      Method: void Increment(int i)
25      precondition: ((0 < i) && (i <= 5))
26      postcondition:  0 <= val <= 50
27     ******************************************/
28     override public void Increment(int i){
29       Debug.Assert((0 < i) && (i <= 5)); // enforce precondition
30       base.Increment(i);
31       if((val+i) <= 50){
32          val += i;
33       }else{
34         int temp = val;
35         temp += i;
36         val = (temp - 50);
37        }
38        CheckInvariant();  // check invariant
39        Console.WriteLine("DerivedIncrementer value is: " + val);
40     }
41
42     private void CheckInvariant(){
43       Debug.Assert((0 <= val) && (val <= 50));
44     }
45   } // end DerivedIncrementer class definition
```

Referring to Example 23.3 — 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 23.4 shows the DerivedIncrementer class in action.

© 2008 Rick Miller — All Rights Reserved

23.4 MainTestApp.cs (Mod 1)

```
1    using System;
2
3    public class MainTestApp {
4      public static void Main(){
5        Incrementer i1 = new Incrementer(0);
6        Incrementer i2 = new DerivedIncrementer(20);
7        i1.Increment(1);
8        i1.Increment(2);
9        i1.Increment(3);
10       i1.Increment(4);
11       i1.Increment(5);
12       Console.WriteLine("-----------------------------");
13       i2.Increment(4);
14       i2.Increment(5);
15       i2.Increment(6); // will cause an assertion error
16     } // end main() method
17   }// end MainTestApp clas definition
```

Referring to Example 23.4 — an Incrementer type reference named i2 is declared on line 5 and initialized to point to an object of type DerivedIncrementer. From lines 13 through 15 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 assertion fails as expected. Figure 23-2 shows the results of running this program.

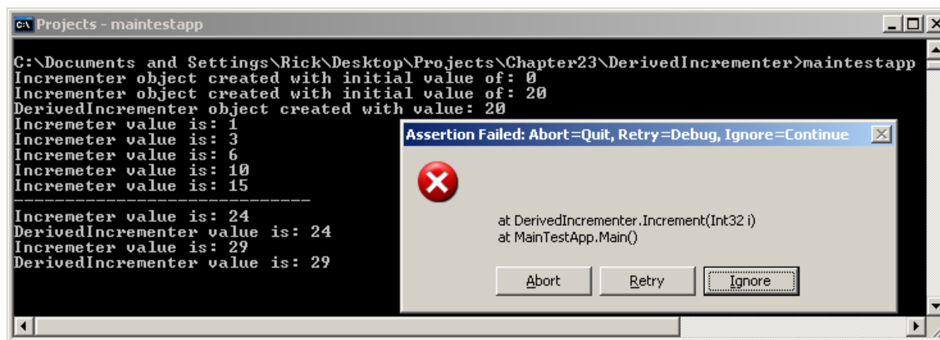


Figure 23-2: Results of Running Example 23.4

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

 © 2008 Rick Miller — All Rights Reserved

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 23.5.

23.5 WeakenedDerivedIncrementer.cs

```
1    #define DEBUG
2    using System;
3    using System.Diagnostics;
4
5    public class WeakenedDerivedIncrementer : Incrementer {
6      /*******************************************
7        Class invariant: 0 <= val <= 50
8      ****************************************/
9      private int val = 0;
10
11     /******************************************************
12       Constructor Method: WeakenedDerivedIncrementer(int i)
13             precondition: ((0 <= i) && (i <= 50))
14            postcondition: 0 <= val <= 50
15      ****************************************************/
16     public WeakenedDerivedIncrementer(int i):base(i){
17       Debug.Assert((0 <= i) && (i <= 50));  // enforce precondition
18       val = i;
19       Console.WriteLine("WeakenedDerivedIncrementer object created with value: " + val);
20       CheckInvariant();
21     }
22
23     /**********************************************
24              Method: void Increment(int i)
25        precondition: ((0 < i) && (i <= 10))
26       postcondition: 0 <= val <= 50
27     *********************************************/
28     override public void Increment(int i){
29       Debug.Assert((0 < i) && (i <= 10)); // enforce precondition
30
31       if((0 <= i) && (i <= 5)){  // remember, it's our job to use the base class correctly!
32            base.Increment(i);
33        }
34
35       if((val+i) <= 50){
36          val += i;
37       }else{
38         int temp = val;
39         temp += i;
40         val = (temp - 50);
41        }
42        CheckInvariant();  // check invariant
43        Console.WriteLine("WeakenedDerivedIncrementer value is: " + val);
44     }
45     /*******************************************
46         Method: void CheckInvariant() - called
47         immediately after any change to class
48         invariant to ensure invariant condition
49         is satisfied.
50       *****************************************/
51     private void CheckInvariant(){
52       Debug.Assert((0 <= val) && (val <= 50));
53     }
54   } // end WeakenedDerivedIncrementer class definition
```

Referring to Example 23.5 — 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 31 ensures the value of i used in the call to the base class version of Increment() obeys its precondition. Example 23.6 shows the WeakenedDerivedIncrementer class being put through its paces in a modified version of the MainTestApp program.

23.6 MainTestApp.cs (Mod 2)

```
1    using System;
2
3    public class MainTestApp {
4      public static void Main(){
5        Incrementer i1 = new Incrementer(0);
6        Incrementer i2 = new DerivedIncrementer(20);
7        Incrementer i3 = new WeakenedDerivedIncrementer(10);
8        i1.Increment(1);
9        i1.Increment(2);
10       i1.Increment(3);
11       i1.Increment(4);
```


© 2008 Rick Miller — All Rights Reserved

```
12        i1.Increment(5);
13        Console.WriteLine("-----------------------------");
14        i2.Increment(4);
15        i2.Increment(5);
16        Console.WriteLine("-----------------------------");
17        i3.Increment(5);
18        i3.Increment(6); // it does not cause an error here...
19        i3.Increment(7); // nor here
20        i3.Increment(8); // nor here
21        i3.Increment(9); // nor here
22        i3.Increment(10); // nor here
23        i3.Increment(11); // ...but here it does!
24     } // end main() method
25    }// end MainTestApp clas definition
```

Referring to Example 23.6 — a new Incrementer type reference named i3 is declared and initialized to point to an object of type WeakenedDerivedIncrementer. Lines 17 through 23 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 assertion error would have been occured on line 18.

However, from the point of view of a programmer who is expecting derived class objects to fulfill the contract of the base class Increment() method, WeakenedDerivedIncrementer objects work just fine because they allow the valid increment ranges of 1 through 5, which is exactly what Incrementer.Increment() methods expect. Figure 23-3 shows the results of running this modified version of MainTestApp.

Figure 23-3: Results of Running Example 23.6

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 then 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 23.7 gives the code for a class named StrengthenedDerivedIncrementer whose Increment() method overrides the base class version and strengthens the precondition.

© 2008 Rick Miller — All Rights Reserved

23.7 StrengthenedDerivedIncrementer.cs

```
1    #define DEBUG
2    using System;
3    using System.Diagnostics;
4
5    public class StrengthenedDerivedIncrementer : Incrementer {
6      /*****************************************
7        Class invariant: 0 <= val <= 50
8      *****************************************/
9      private int val = 0;
10
11     /*******************************************************
12       Constructor Method: StrengthenedDerivedIncrementer(int i)
13             precondition: ((0 <= i) && (i <= 50))
14            postcondition: 0 <= val <= 50
15      *****************************************************/
16     public StrengthenedDerivedIncrementer(int i):base(i){
17       Debug.Assert((0 <= i) && (i <= 50));  // enforce precondition
18       val = i;
19       Console.WriteLine("StrengthenedDerivedIncrementer object created with value: " + val);
20       CheckInvariant();
21     }
22
23     /***************************************************
24              Method: void Increment(int i)
25        precondition: ((0 < i) && (i <= 3))
26       postcondition: 0 <= val <= 50
27     ***************************************************/
28     override public void Increment(int i){
29       Debug.Assert((0 < i) && (i <= 3)); // enforce precondition
30       base.Increment(i);
31       if((val+i) <= 50){
32          val += i;
33       }else{
34         int temp = val;
35         temp += i;
36         val = (temp - 50);
37        }
38        CheckInvariant();  // check invariant
39        Console.WriteLine("StrengthenedDerivedIncrementer value is: " + val);
40     }
41     /*******************************************
42         Method: void CheckInvariant() - called
43         immediately after any change to class
44         invariant to ensure invariant condition
45         is satisfied.
46       *****************************************/
47     private void CheckInvariant(){
48       Debug.Assert((0 <= val) && (val <= 50));
49     }
50
51   } // end StrengthenedDerivedIncrementer class definition
```

Referring to Example 23.7 — the StrengthenedDerivedIncrementer class places a restriction on the original Increment() method precondition by limiting the authorized increment values to 1 through 3. Example 23.8 shows the StrengthenedDerivedIncrementer class in action. Figure 23-4 shows the results of running this program.

23.8 MainTestApp.cs (Mod 3)

```
1    using System;
2
3    public class MainTestApp {
4      public static void Main(){
5        Incrementer i1 = new Incrementer(0);
6        Incrementer i2 = new DerivedIncrementer(20);
7        Incrementer i3 = new WeakenedDerivedIncrementer(10);
8        Incrementer i4 = new StrengthenedDerivedIncrementer(10);
9
10       i1.Increment(1);
11       i1.Increment(2);
12       i1.Increment(3);
13       i1.Increment(4);
14       i1.Increment(5);
15       Console.WriteLine("-----------------------------");
16       i2.Increment(4);
17       i2.Increment(5);
18       Console.WriteLine("-----------------------------");
19       i3.Increment(5);
20       Console.WriteLine("-----------------------------");
21       i4.Increment(2); // OK so far...
22       i4.Increment(3); // OK here too...
23       i4.Increment(4); // Wait a minute...this should work!
```

 © 2008 Rick Miller — All Rights Reserved

```
24
25    } // end main() method
26   }// end MainTestApp clas definition
```

```
Projects - maintestapp
C:\Documents and Settings\Rick\Desktop\Projects\Chapter23\StrengthenedIncrementer>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
Incremeter value is: 1
Incremeter value is: 3
Incremeter value is: 6
Incremeter value is: 10
Incremeter value is: 15
------------------------------
Incremeter value is: 24
DerivedIncrementer value is: 24
Incremeter value is: 29
DerivedIncrementer value is: 29
------------------------------
Incremeter value is: 15
WeakenedDerivedIncrementer value is: 15
------------------------------
Incremeter value is: 12
StrengthenedDerivedIncrementer value is: 12
Incremeter value is: 15
StrengthenedDerivedIncrementer value is: 15

Assertion Failed: Abort=Quit, Retry=Debug, Ignore=Continue
at StrengthenedDerivedIncrementer.Increment(Int32 i)
at MainTestApp.Main()
Abort  Retry  Ignore
```

Figure 23-4: Results of Running Example 24.8

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.

The Incrementer and its derived class examples shown previously each had their own private attribute that was part if 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.

 © 2008 Rick Miller — All Rights Reserved

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 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 23-5.

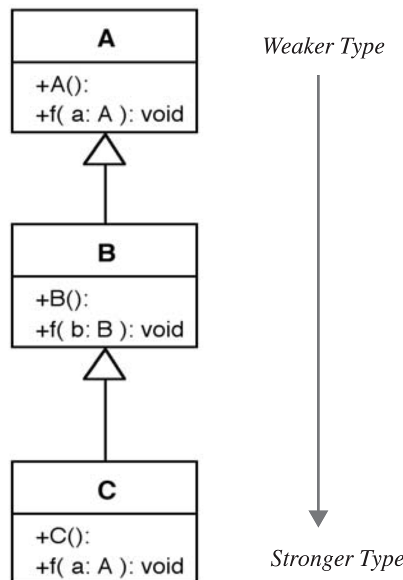


Figure 23-5: Strong vs. Weak Types

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 23.9 through 23.11 give the code for classes A, B, and C. Example 23.12 puts these classes through their paces, and Figure 23-6 shows the results of running this program.

23.9 A.cs

```
1    using System;
2
3    public class A {
4      public A(){
5         Console.WriteLine("A object created!");
6      }
7
8      public virtual void f(A a){
9        Console.WriteLine("A.f() called!");
10     }
11   }
```

23.10 B.cs

```
1    using System;
2
3    public class B : A {
4      public B(){
```

 © 2008 Rick Miller — All Rights Reserved

```
5         Console.WriteLine("B object created!");
6       }
7
8       public virtual void f(B b){
9         Console.WriteLine("B.f() called!");
10      }
11    }
```

23.11 C.cs

```
1     using System;
2
3     public class C : B {
4       public C(){
5          Console.WriteLine("C object created!");
6       }
7
8       public override void f(A a){
9         Console.WriteLine("C.f() called!");
10      }
11    }
```

23.12 MainApp.cs

```
1     using System;
2
3     public class MainApp {
4       public static void Main(){
5         A a1 = new A();
6         a1.f(new A()); // A's method called
7
8         Console.WriteLine("------------------");
9
10        A a2 = new B();
11        a2.f(new A()); // A's method called
12        a2.f(new B()); // A's method called
13
14        Console.WriteLine("------------------");
15
16        B b1 = new C();
17        b1.f(new A());  // C's overriding method called
18        b1.f(new B()); // B's overloaded method called
19        b1.f(new C()); // B's overloaded method called
20
21        Console.WriteLine("------------------");
22
23        A a3 = new C();
24        a3.f(new A()); // C's overriding method called
25        a3.f(new B()); // C's overriding method called
26        a3.f(new C()); // C's overriding method called
27
28      } // end Main() method
29    } // end MainApp program
```

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 23-5. 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.

 © 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter23\Strong_Weak_Types>mainapp
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!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter23\Strong_Weak_Types>
```

Figure 23-6: Results of Running Example 24.12

Signature Rule

The signature rule deals with the methods published or made public by a type specification. In C# 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. C# 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.

© 2008 Rick Miller — All Rights Reserved

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 is because abstractions tend to be more stable. (Correctly designed abstractions are very stable!) This is achieved in C# 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 23.13 through 23.24. 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 23-7 gives the UML diagram for the naval fleet class inheritance hierarchy. Example 23.24 offers a short program showing the naval fleet classes in action, and Figure 23-8 shows the results of running this program.

23.13 Vessel.cs

```
1    using System;
2
3    public abstract class Vessel {
4      private Plant its_plant = null;
5      private Weapon its_weapon = null;
6      private String its_name = null;
7
8      // protected properties
9      protected Weapon Weapon {
10       get { return its_weapon; }
11     }
12
13
14     protected Plant Plant {
15       get { return its_plant; }
16     }
```

 © 2008 Rick Miller — All Rights Reserved

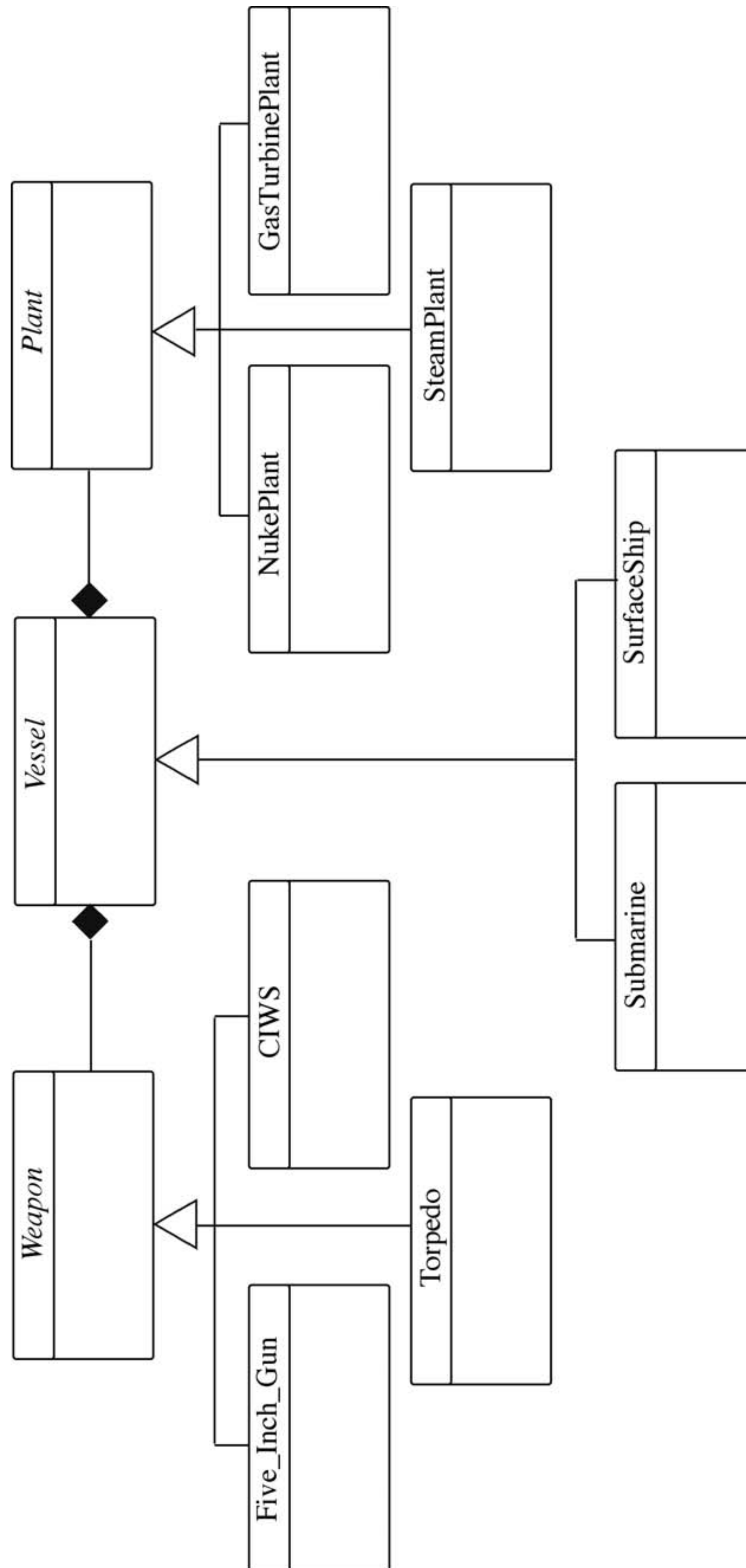


Figure 23-7: Naval Fleet Class Inheritance Hierarchy

© 2008 Rick Miller — All Rights Reserved

```
17
18
19      public Vessel(Plant plant, Weapon weapon, String name){
20        its_weapon = weapon;
21        its_plant = plant;
22        its_name = name;
23        Console.WriteLine("The vessel " + its_name + " created!");
24      }
25
26      /* ******************************************************
27          Public Abstract Methods - must be implemented in
28          derived classes.
29      ******************************************************/
30      public abstract void LightoffPlant();
31      public abstract void ShutdownPlant();
32      public abstract void TrainWeapon();
33      public abstract void FireWeapon();
34
35      /* ******************************************************
36          ToString() Method - may be overridden in subclasses.
37      ******************************************************/
38      public override String ToString(){
39        return "Vessel name: " + its_name + " " + its_plant.ToString() +
40                " " + its_weapon.ToString();
41      }
42
43
44    }// end Vessel class definition
```

23.14 Plant.cs

```
1     using System;
2
3     public abstract class Plant {
4        private String its_model = null;
5        public Plant(String model){
6          its_model = model;
7        }
8        public abstract void LightoffPlant();
9        public abstract void ShutdownPlant();
10
11       public override String ToString(){ return "Plant model: " + its_model; }
12    }
```

23.15 Weapon.cs

```
1     using System;
2
3     public abstract class Weapon {
4       private String its_model = null;
5
6       public Weapon(String model){
7         its_model = model;
8         Console.WriteLine("Weapon object created!");
9       }
10
11      public abstract void TrainWeapon();
12      public abstract void FireWeapon();
13
14      public override String ToString(){ return "Weapon model: " + its_model; }
15    }
```

23.16 CIWS.cs

```
1     using System;
2
3     public class CIWS : Weapon {
4
5        public CIWS(String model):base(model){
6         Console.WriteLine("CIWS object created!");
7        }
8
9        public override void TrainWeapon(){
10        Console.WriteLine("CIWS is locked on target!");
11       }
12
13       public override void FireWeapon(){
14        Console.WriteLine("The CIWS roars to life and fires a zillion bullets at the target!");
```


© 2008 Rick Miller — All Rights Reserved

```
15      }
16   }
```

23.17 Torpedo.cs

```
1    using System;
2
3    public class Torpedo : Weapon {
4
5       public Torpedo(String model):base(model) {
6        Console.WriteLine("Torpedo object created!");
7       }
8
9       public override void TrainWeapon(){
10       Console.WriteLine("Torpedo is locked on target!");
11      }
12
13      public override void FireWeapon(){
14       Console.WriteLine("Fish in the water, heading towards target!");
15      }
16   }
```

23.18 Five_Inch_Gun.cs

```
1    using System;
2
3    public class Five_Inch_Gun : Weapon {
4
5       public Five_Inch_Gun(String model):base(model){
6        Console.WriteLine("Five Inch Gun object created!");
7       }
8
9       public override void TrainWeapon(){
10       Console.WriteLine("Five Inch Gun is locked on target!");
11      }
12
13      public override void FireWeapon(){
14       Console.WriteLine("Blam! Blam! Blam!");
15      }
16   }
```

23.19 SteamPlant.cs

```
1    using System;
2
3    public class SteamPlant : Plant {
4
5       public SteamPlant(String model):base(model) {
6         Console.WriteLine("SteamPlant object created!");
7       }
8
9      public override void LightoffPlant(){
10       Console.WriteLine("Steam pressure is rising!");
11     }
12
13     public override void ShutdownPlant(){
14       Console.WriteLine("Steam plant is secure!");
15     }
16   }
```

23.20 NukePlant.cs

```
1    using System;
2
3    public class NukePlant : Plant {
4
5       public NukePlant(String model):base(model) {
6         Console.WriteLine("NukePlant object created!");
7       }
8
9      public override void LightoffPlant(){
10       Console.WriteLine("Nuke plant is critical!");
11     }
12
13     public override void ShutdownPlant(){
14       Console.WriteLine("Nuke plant is secure!");
```


© 2008 Rick Miller — All Rights Reserved

```
15      }
16    }
```

23.21 GasTurbinePlant.cs

```
1     using System;
2
3     public class GasTurbinePlant : Plant {
4
5        public GasTurbinePlant(String model):base(model) {
6          Console.WriteLine("GasTurbinePlant object created!");
7        }
8
9       public override void LightoffPlant(){
10        Console.WriteLine("Gas Turbine is running and ready to go!");
11      }
12
13      public override void ShutdownPlant(){
14        Console.WriteLine("Gas Turbine is secure!");
15      }
16    }
```

23.22 Submarine.cs

```
1     using System;
2
3     public class Submarine : Vessel {
4
5        public Submarine(Plant plant, Weapon weapon, String name):base(plant, weapon, name){
6          Console.WriteLine("Submarine object created: " + base.ToString());
7        }
8
9        public override void LightoffPlant(){
10         Plant.LightoffPlant();
11       }
12
13       public override void ShutdownPlant(){
14         Plant.ShutdownPlant();
15       }
16
17       public override void TrainWeapon(){
18         Weapon.TrainWeapon();
19       }
20
21       public override void FireWeapon(){
22         Weapon.FireWeapon();
23       }
24
25    } // end Submarine class definition
```

23.23 SurfaceShip.cs

```
1     using System;
2
3     public class SurfaceShip : Vessel {
4
5        public SurfaceShip(Plant plant, Weapon weapon, String name):base(plant, weapon, name){
6          Console.WriteLine("SurfaceShip object created: " + base.ToString());
7        }
8
9        public override void LightoffPlant(){
10         Plant.LightoffPlant();
11       }
12
13       public override void ShutdownPlant(){
14         Plant.ShutdownPlant();
15       }
16
17       public override void TrainWeapon(){
18         Weapon.TrainWeapon();
19       }
20
21       public override void FireWeapon(){
22         Weapon.FireWeapon();
23       }
24
25    } // end SurfaceShip class definition
```

23.24 FleetTestApp.cs

```
1     using System;
2
```


© 2008 Rick Miller — All Rights Reserved

```
3    public class FleetTestApp {
4      public static void Main(){
5        Vessel v1 = new Submarine(new NukePlant("Preasureized Water Mk 85"), new Torpedo("MK 50"),
6                                   "USS Falls Church");
7        v1.LightoffPlant();
8        v1.TrainWeapon();
9        v1.FireWeapon();
10       v1.ShutdownPlant();
11     }
12   }// end FleetTestApp class definition
```

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter23\Fleet_Sim>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: Preasureized Water Mk 85 Weapon model: MK 50
Nuke plant is critical!
Torpedo is locked on target!
Fish in the water, heading towards target!
Nuke plant is secure!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter23\Fleet_Sim>_
```

Figure 23-8: Results of Running Example 24.22

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 C# 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 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 23-9 where high-level modules depend on low-level modules. Referring to Figure 23-9 — 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*.

© 2008 Rick Miller — All Rights Reserved

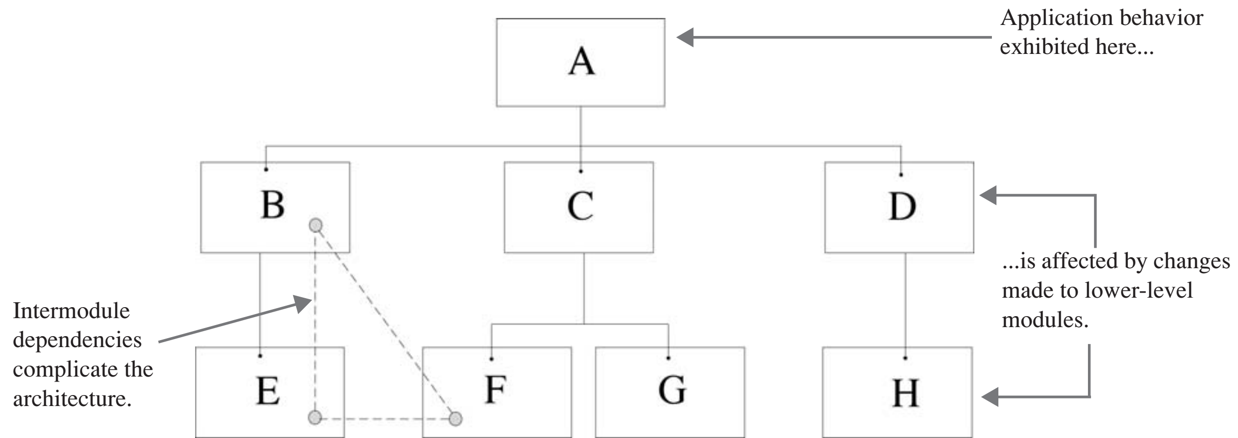


Figure 23-9: Traditional Top-Down Functional Dependencies

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 software modules for reuse in other systems. A 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 it 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 23-7. 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).

 © 2008 Rick Miller — All Rights Reserved

Terms and Definitions

The terms and definitions listed in Table 23-1 were used throughout this chapter:

Term	Definition
Abstraction	The separation of the important from the unimportant. (*i.e.*, interface vs. implementation)
Abstract Data Type	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.
Supertype	An abstract data type that serves as a specification for related subtypes.
Subtype	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 and then add specialized behavior if required.
Type Specification	A declaration of the behavioral properties of an abstract data type. A specification describes the important characteristics of the data abstraction.
Encapsulation	The act of hiding private implementation details behind a publicly accessible interface.
Precondition	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.
Postcondition	A condition, constraint, or set of constraints that must be satisfied when an abstract data type method completes execution.
Inheritance Hierarchy	A set of abstract data type specifications that implement a supertype and subtype relationship between each abstract data type.
Class	The declaration of an abstract data type specifying a set of attributes and interface methods common to a set of objects.
Abstract Class	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.
Subclass	A declaration of an abstract data type taking all or part of its specification from another, possibly abstract, class.
Class Invariant	An assertion about the state of an object which must hold true for all possible states the object may assume.

Table 23-1: Terms and Definitions Used in this Chapter

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

 © 2008 Rick Miller — All Rights Reserved

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 C# 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 C# 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. **API Drill:** Study the System.Diagnostics namespace. List each class and write a brief description about each one.

Suggested Projects

1. **Robot Rat:** Evaluate your latest version of Robot Rat 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.

© 2008 Rick Miller — All Rights Reserved

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

© 2008 Rick Miller — All Rights Reserved

Notes

© 2008 Rick Miller — All Rights Reserved

Chapter 24

Nikon F3HP / Zoom-Nikkor 35-105 / Kodak Tri-X

Line Handlers Assemble

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

© 2008 Rick Miller — All Rights Reserved

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 C# .NET. 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 use them 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 between 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

© 2008 Rick Miller — All Rights Reserved

the imperfect design to an implementation results in an imperfect implementation, and 5) C# and the .NET Framework both suffer from imperfection, as do all object-oriented programming languages and frameworks humans create. Therefore — to argue perfect design is a waste of time.

Design pragmatists will attempt to achieve reuse at very 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.

© 2008 Rick Miller — All Rights Reserved

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 .NET Framework 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.

© 2008 Rick Miller — All Rights Reserved

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 20)

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 24-1. Table 24-1 provides a brief description of each inheritance form. (**Note:** The lightly shaded rows of Table 24-1 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.

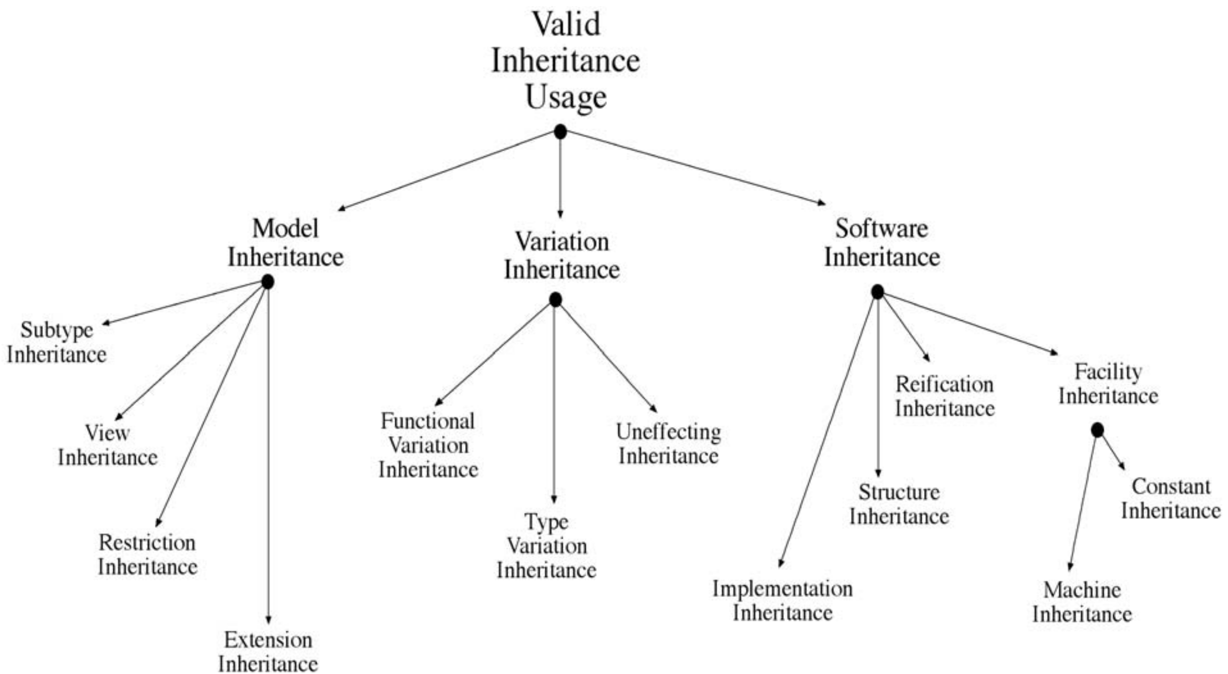


Figure 24-1: Meyer's Inheritance Taxonomy

Inheritance Form	Inheritance Form	Inheritance Form Or Description
Model	Subtype	The most obvious form of inheritance. Used to model application domain objects into categories (base classes) and subcategories (derived classes). Base classes serve to specify behavior only and are therefore abstract. (*Or, in C#, an interface.*) 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 *invariant*. (An invariant is a property that must hold true at all times. I discussed class invariants in detail in Chapter 23.)
	Extension	Derived classes introduce new behavior not found in the base class.

Table 24-1: Inheritance Form Descriptions

© 2008 Rick Miller — All Rights Reserved

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 (or interfaces).	
Variation	Functional Variation	Derived classes redefine (override) base class methods.	
	Type Variation	Derived classes redefine base class method signatures. This type of inheritance is not authorized in C#. 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 (non-abstract).	
	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 (abstract methods). 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 .NET 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 const 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 24-1: 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.*

© 2008 Rick Miller — All Rights Reserved

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 24-2 gives the UML class diagram.

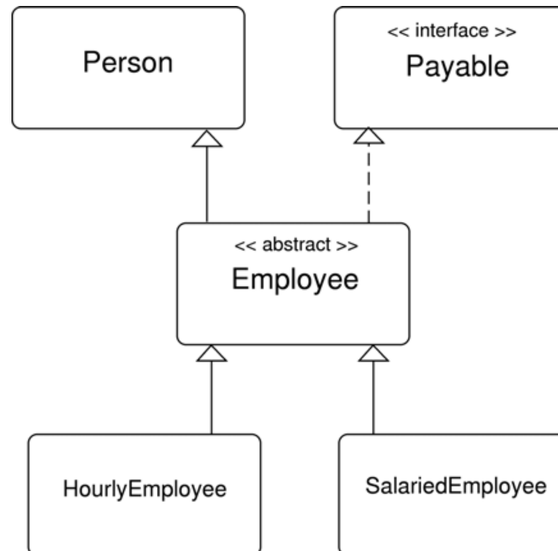


Figure 24-2: Person-Employee Inheritance Diagram

Referring to Figure 24-2 — 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 a few 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.

 © 2008 Rick Miller — All Rights Reserved

The Role Of Interfaces

The interface construct provides the means to specify type behavior. Interfaces can inherit from other interfaces. This facilitates the application of a rich variety of inheritance forms to include: *subtype inheritance* and *extension inheritance*.

A class can inherit from only one other class, 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 IComparable<T> by implementing the IComparable<T> interface and providing behavior for its CompareTo() 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 herding 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 list 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

 © 2008 Rick Miller — All Rights Reserved

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. Examples of collateral roles abound in the .NET Framework. For example, the DateTime structure whose primary role is a value type that stores date and time values can be compared with other DateTime object because it implements the IComparable interface, it also implements IFormattable, IConvertible, and ISerializable.

Dynamic Roles

In C#, 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 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 24-2. 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 Hour-

© 2008 Rick Miller — All Rights Reserved

lyEmployee 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 chose 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.

 © 2008 Rick Miller — All Rights Reserved

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 24-3 gives the class diagram for the complete application.

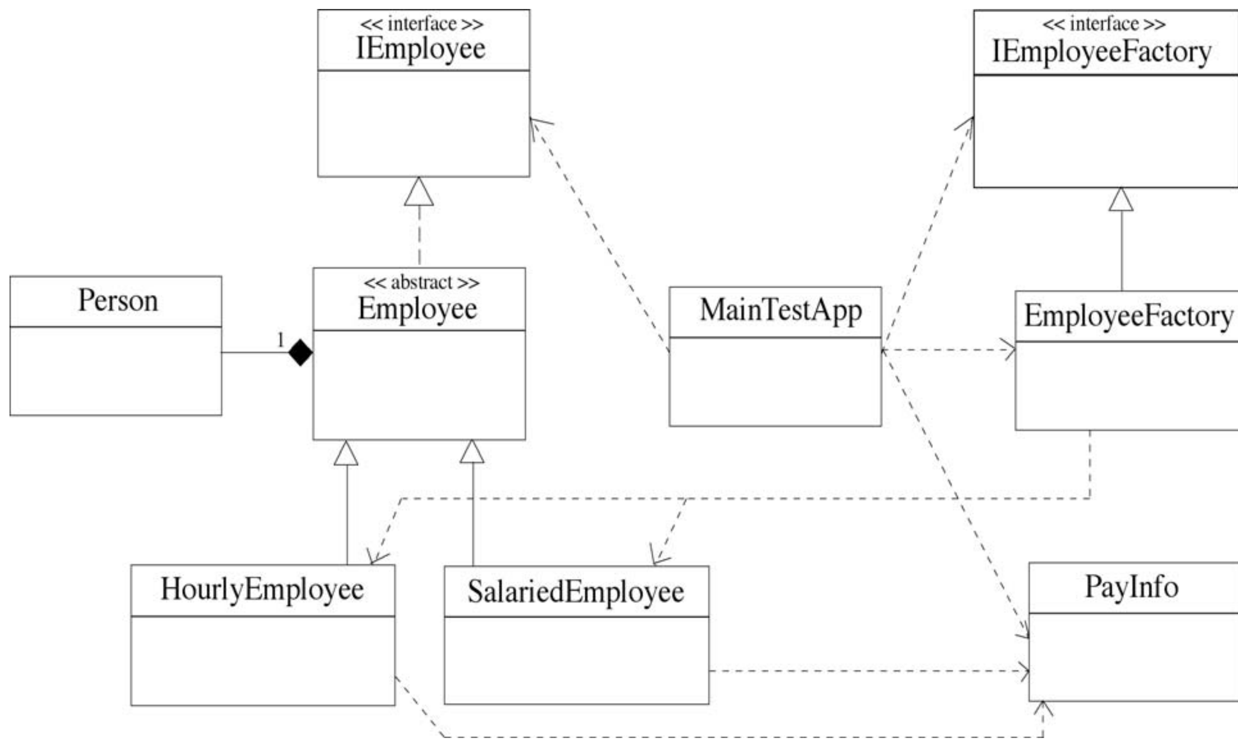


Figure 24-3: Revised Person - Employee Example

Referring to Figure 24-3 — 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 may have been 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 24.1 through 24.10 provide the code for this application.

 © 2008 Rick Miller — All Rights Reserved

24.1 IEmployee.cs

```
1    using System;
2
3      public interface IEmployee : IComparable<IEmployee> {
4         int Age { get; }
5         String FullName { get; }
6         String FullNameAndAge { get; }
7         String FirstName { get; set; }
8         String MiddleName { get; set; }
9         String LastName { get; set; }
10        String EmployeeNumber { get; set; }
11        DateTime Birthday { get; set; }
12        Sex Gender { get; set; }
13        PayInfo PayInfo { set; get; }
14        double Pay { get; }
15     }
16   }
```

24.2 Employee.cs

```
1    using System;
2
3      [Serializable]
4      public abstract class Employee : IEmployee {
5         private Person _person = null;
6         private String  _employee_number = null;
7         private PayInfo _payInfo = null;
8
9         protected Employee(){
10          _person = new Person();
11        }
12
13        protected Employee(String f_name, String m_name, String l_name, Sex gender, DateTime birthday,
14                           String employee_number){
15           _person = new Person(f_name, m_name, l_name, gender, birthday );
16           _employee_number = employee_number;
17         } // end constructor
18
19        public int Age {
20          get{ return _person.Age; }
21         }
22        public String FullName {
23           get { return _person.FullName; }
24        }
25        public String FullNameAndAge {
26          get { return _person.FullNameAndAge; }
27        }
28        public String FirstName {
29          get { return _person.FirstName; }
30          set { _person.FirstName = value; }
31       }
32        public String MiddleName {
33          get { return _person.MiddleName; }
34          set { _person.MiddleName = value; }
35        }
36        public String LastName {
37          get { return _person.LastName; }
38          set { _person.LastName = value; }
39        }
40        public Sex Gender {
41          get { return _person.Gender; }
42          set { _person.Gender = value; }
43        }
44        public String EmployeeNumber {
45           get { return _employee_number; }
46           set { _employee_number = value; }
47        }
48        public DateTime Birthday {
49          get { return _person.BirthDay; }
50          set { _person.BirthDay = value; }
51        }
52
53        public PayInfo PayInfo {
54          get { return _payInfo; }
55          set { _payInfo = value; }
56        }
57
58        // defer implementation of this property
59        public abstract double Pay { get; }
60
61        public override String ToString(){
```

 © 2008 Rick Miller — All Rights Reserved

```
62          return (_person.ToString() + " " + EmployeeNumber + " ");
63        }
64
65        public int CompareTo(IEmployee other){
66          return this.ToString().CompareTo(other.ToString());
67        }
68     }  // end Employee class definition
```

24.3 Person.cs

```
1    using System;
2
3      [Serializable]
4      public class Person : IComparable<Person> {
5
6        // private instance fields
7        private String   _firstName;
8        private String   _middleName;
9        private String   _lastName;
10       private Sex      _gender;
11       private DateTime _birthday;
12
13
14       //private default constructor
15       public Person(){}
16
17       public Person(String firstName, String middleName, String lastName,
18                     Sex gender, DateTime birthday){
19          FirstName = firstName;
20          MiddleName = middleName;
21          LastName = lastName;
22          Gender = gender;
23          BirthDay = birthday;
24       }
25
26       // public properties
27       public String FirstName {
28         get { return _firstName; }
29         set { _firstName = value; }
30       }
31
32       public String MiddleName {
33         get { return _middleName; }
34         set { _middleName = value; }
35       }
36
37       public String LastName {
38         get { return _lastName; }
39         set { _lastName = value; }
40       }
41
42       public Sex Gender {
43         get { return _gender; }
44         set { _gender = value; }
45       }
46
47       public DateTime BirthDay {
48         get { return _birthday; }
49         set { _birthday = value; }
50       }
51
52      public int Age {
53        get {
54         int years = DateTime.Now.Year - _birthday.Year;
55         int adjustment = 0;
56           if(DateTime.Now.Month < _birthday.Month){
57             adjustment = 1;
58           }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
59                     adjustment = 1;
60                 }
61          return years - adjustment;
62        }
63      }
64
65       public String FullName {
66         get { return FirstName + " " + MiddleName + " " + LastName; }
67       }
68
69       public String FullNameAndAge {
70         get { return FullName + " " + Age; }
71       }
```


© 2008 Rick Miller — All Rights Reserved

```
72
73      public override String ToString(){
74        return FullName + ", " + Gender + ", " + Age;
75      }
76
77      public int CompareTo(Person other){
78        return this.FullName.CompareTo(other.FullName);
79      }
80
81    } // end Person class
```

24.4 SexEnum.cs

```
1    using System;
2
3    [Serializable]
4    public enum Sex {MALE, FEMALE}
```

24.5 HourlyEmployee.cs

```
1    using System;
2
3      [Serializable]
4      public class HourlyEmployee : Employee {
5
6        public HourlyEmployee():base() { }
7
8        public HourlyEmployee(String f_name, String m_name, String l_name, Sex gender, DateTime birthday,
9                              String employee_number)
10              :base(f_name, m_name, l_name, gender, birthday,  employee_number){ }
11
12        public override double Pay {
13          get { return base.PayInfo.HoursWorked * base.PayInfo.HourlyRate; }
14        }
15
16        public override String ToString() {
17          return (base.ToString() + " " + Pay.ToString("C3"));
18
19        }
20     } // end HourlyEmployee class definition
```

24.6 SalariedEmployee.cs

```
1    using System;
2
3      [Serializable]
4      public class SalariedEmployee : Employee {
5
6         public SalariedEmployee():base() { }
7
8         public SalariedEmployee(String f_name, String m_name, String l_name, Sex gender, DateTime birthday,
9                                String employee_number)
10         :base(f_name, m_name, l_name, gender, birthday, employee_number){ }
11
12        public override double Pay {
13          get { return ((base.PayInfo.Salary/12.0)/2.0); }
14        }
15
16        public override String ToString() {
17          return (base.ToString() + " " + Pay.ToString("C3"));
18        }
19     } // end SalariedEmployee class definition
```

24.7 PayInfo.cs

```
1    using System;
2
3       [Serializable]
4      public class PayInfo {
5        // fields
6        private double _salary = 0;
7        private double _hours_worked = 0;
8        private double _hourly_rate = 0;
9
10       // properties
11       public double Salary {
12          get { return _salary; }
13          set { _salary = value; }
14       }
15
16       public double HoursWorked {
17         get { return _hours_worked; }
```


© 2008 Rick Miller — All Rights Reserved

```
18        set { _hours_worked = value; }
19      }
20
21      public double HourlyRate {
22        get { return _hourly_rate; }
23        set { _hourly_rate = value; }
24      }
25
26      // constructors
27      public PayInfo(){ }
28
29      public PayInfo(double salary){
30         _salary = salary;
31      }
32      public PayInfo(double hours_worked, double hourly_rate){
33         _hours_worked = hours_worked;
34         _hourly_rate = hourly_rate;
35      }
36    } // end PayInfo class definition
```

24.8 IEmployeeFactory.cs

```
1   using System;
2
3     public interface IEmployeeFactory {
4       IEmployee GetNewSalariedEmployee(String f_name, String m_name, String l_name,Sex gender,
5                                        DateTime birthday, String employee_number);
6       IEmployee GetNewHourlyEmployee(String f_name, String m_name, String l_name, Sex gender,
7                                      DateTime birthday, String employee_number);
8     }
```

24.9 EmployeeFactory.cs

```
1   using System;
2
3     public class EmployeeFactory : IEmployeeFactory {
4
5       public IEmployee GetNewSalariedEmployee(String f_name, String m_name, String l_name, Sex gender,
6                                                DateTime birthday, String employee_number){
7         return new SalariedEmployee(f_name, m_name, l_name, gender, birthday, employee_number);
8       }
9
10      public IEmployee GetNewHourlyEmployee(String f_name, String m_name, String l_name, Sex gender,
11                                              DateTime birthday, String employee_number){
12         return new HourlyEmployee(f_name, m_name, l_name, gender, birthday,  employee_number);
13      }
14    } // end EmployeeFactory class definition
```

24.10 MainTestApp.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   public class MainTestApp {
5
6       private IEmployeeFactory _employee_factory = null;
7       private List<IEmployee> _employee_list = null;
8
9       public MainTestApp(){
10        _employee_factory = new EmployeeFactory();
11        _employee_list = new List<IEmployee>();
12      }
13
14      public void CreateEmployees(){
15        _employee_list.Add(_employee_factory.GetNewSalariedEmployee("Rick", "Warren", "Miller",
16                                                                     Sex.MALE, new DateTime(1968, 2, 4),
17                                                                     "0001"));
18         _employee_list[0].PayInfo = new PayInfo(78000);
19        _employee_list.Add(_employee_factory.GetNewHourlyEmployee("Coralie", "Sylvia", "Powell",
20                                                                     Sex.FEMALE, new DateTime(1969, 4, 8),
21                                                                     "0002"));
22         _employee_list[1].PayInfo = new PayInfo(80, 57);
23      }
24
25      public void ListEmployees(){
26        foreach(IEmployee e in _employee_list){
27          Console.WriteLine(e);
28        }
29      }
30
31      public static void Main(){
```


© 2008 Rick Miller — All Rights Reserved

```
32          MainTestApp mta = new MainTestApp();
33          mta.CreateEmployees();
34          mta._employee_list.Sort();
35          mta.ListEmployees();
36        } // end Main
37    } // end class definition
```

Referring to Example 24.10 — the MainTestApp has two private fields: one of type IEmployeeFactory and the other a generic list of type IEmployee. The constructor method on line 9 initializes the _employee_factory reference to point to an EmployeeFactory instance. The constructor also initializes the _employee_list reference.

The CreateEmployees() method beginning on line 14 uses the _employee_factory reference to create an HourlyEmployee and a SalariedEmployee, adding each to the _employee_list. (Remember, the EmployeeFactory returns references to objects that implement the IEmployee interface.) The CreateEmployees() method then sets each employee's pay information by assigning an appropriately initialized PayInfo object to the PayInfo property.

The ListEmployees() method beginning on line 25 simply iterates through the _employee_list and prints information about each employee to the console.

The Main() method starts on line 31 and creates an instance of the MainTestApp class followed by a call to the CreateEmployees() and ListEmployees() methods respectively. In between these method calls it sorts the list by calling the _employee_list.Sort() method. Figure 24-4 shows the results of running this program.

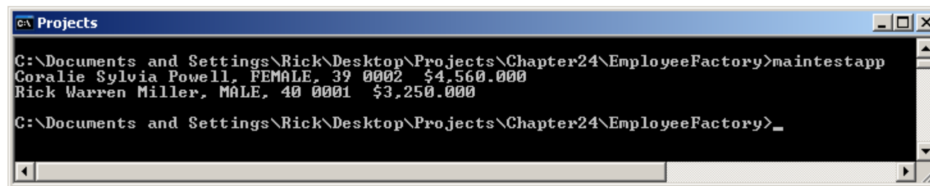


Figure 24-4: Results of Running Example 24.9

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 chose 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.

 © 2008 Rick Miller — All Rights Reserved

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 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 chose 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 24.10.

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.

© 2008 Rick Miller — All Rights Reserved

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 : 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 virtual void f(){ Console.WriteLine("Hello from BaseClass f()!"); }
3    }
```

```
1    public class DerivedClass : BaseClass {
2      public override void f() { Console.WriteLine("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. **Robot Rat Encore Une Fois:** Revisit the Robot Rat 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 multithreaded client-server application.

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 subsystem (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, keypad 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.

© 2008 Rick Miller — All Rights Reserved

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

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

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

Rick Miller. *Java For Artists: The Art, Philosophy, And Science of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-05-2

© 2008 Rick Miller — All Rights Reserved

Notes

© 2008 Rick Miller — All Rights Reserved

Chapter 25

Contax T / Kodak Tri-X

Mayan Ruin – Chichen Itza – Mexico

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 the purpose of the singleton pattern*
- *Describe the purpose of the command pattern*
- *Describe the use of the model-view-controller (MVC) pattern*
- *Combine the MVC, factory, singleton, and command patterns to formulate flexible application architectures*

© 2008 Rick Miller — All Rights Reserved

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 3) 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.*

© 2008 Rick Miller — All Rights Reserved

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	Contents
Pattern Name and Classification	The name of the pattern.
Intent	A description of the pattern's purpose and use.
Also Known As	Other names the pattern may be known by.
Motivation	The problem the pattern is trying to solve.
Applicability	Under what situations the pattern should be applied.
Structure	A graphical representation of the pattern in a notational language like UML.
Participants	The pattern's classes and/or objects and their responsibilities.
Collaborations	How a pattern's participants execute their responsibilities.
Consequences	The ramifications of the pattern's use.
Implementation	Advice on using the pattern.
Sample Code	Concrete pattern implementation example in a programming language like C#.
Known Uses	Examples of the pattern's application in the real world.
Related Patterns	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.

© 2008 Rick Miller — All Rights Reserved

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 configuration objects. The singleton is often used to implement other patterns such as the *factory*. (The factory pattern is discussed in detail later in a separate section.)

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(). The following examples together implement an XML properties class that is used to create, store, and retrieve application properties in an XML file. This example uses the XmlSerializer to persist a list of PropertyEntry objects. (i.e, List<PropertyEntry>). The properties are maintained in a Dictionary<String, String> object but curiously, you cannot use the XmlSerializer to serialize a generic dictionary object. Thus the need to convert the entries in the dictionary into a list of structures which can be serialized to an XML file.

25.1 PropertyEntry.cs

```
1    public struct PropertyEntry {
2      public string PropertyName;
3      public string Value;
4    }
```

Referring to Example 25.1 — this simple structure is all it takes to store individual property entries. You'll see how this structure is used in the next example.

25.2 XMLProperties.cs

```
1    using System;
2    using System.Collections.Generic;
3    using System.IO;
4    using System.Xml;
5    using System.Xml.Serialization;
6
7    public class XMLProperties {
8
9       // private fields
10      private Dictionary<String, String> _properties = null;
11      private String _filename = null;
12
13      // constants
14      protected const String DEFAULT_PROPERTIES_FILENAME = "properties.xml";
15
16      // protected constructors
17      protected XMLProperties():this(DEFAULT_PROPERTIES_FILENAME){ }
18
19      protected XMLProperties(String filename){
20        if((filename == null) || (filename == String.Empty)){
21          _filename = DEFAULT_PROPERTIES_FILENAME;
22        }else {
23          _filename = filename;
24        }
25        _properties = new Dictionary<String, String>();
26
27      }
28
29      /*************************************************************
30        Converts _properties dictionary into List<PropertyEntry> object
31        and serializes it to an xml file.
32     ***********************************************************/
33      public void Store(String filename){
34        TextWriter writer = null;
35        try {
36          writer = new StreamWriter(filename);
37          List<PropertyEntry> entry_list = new List<PropertyEntry>();
38          foreach(KeyValuePair<String, String> entry in _properties){
39            PropertyEntry pe;
40            pe.PropertyName = entry.Key;
41            pe.Value = entry.Value;
42            entry_list.Add(pe);
43          }
44          //remove
45          foreach(PropertyEntry entry in entry_list){
46            Console.WriteLine(entry.PropertyName + ", " + entry.Value);
47          }
48
49          XmlSerializer serializer = new XmlSerializer(typeof(List<PropertyEntry>));
50          serializer.Serialize(writer, entry_list);
51        } catch(IOException ioe){
```

 © 2008 Rick Miller — All Rights Reserved

```
52          Console.WriteLine(ioe);
53        } catch(Exception ex){
54          Console.WriteLine(ex);
55        } finally {
56          if(writer != null) writer.Close();
57        }
58        _filename = filename;
59      }
60
61      /**************************************************************
62        Stores property entries to default file
63      **************************************************************/
64      public void Store(){
65        this.Store(_filename);
66      }
67
68      /**************************************************************
69        Reads the XML properties file and populates the _properties
70        dictionary. Throws an IOException if the specified filename does
71        not exist.
72      *************************************************************/
73      public void Read(String filename ){
74
75        if(!File.Exists(filename)){
76          throw new IOException("Requested file does not exist!");
77        }
78        FileStream fs = null;
79        try {
80          fs = new FileStream(filename, FileMode.Open);
81          XmlSerializer serializer = new XmlSerializer(typeof(List<PropertyEntry>));
82          List<PropertyEntry> entry_list = (List<PropertyEntry>)serializer.Deserialize(fs);
83          foreach(PropertyEntry entry in entry_list){
84            _properties[entry.PropertyName] = entry.Value;
85          }
86
87        }catch(IOException ioe){
88          Console.WriteLine(ioe);
89        }catch(Exception ex){
90          Console.WriteLine(ex);
91        }finally{
92          if(fs != null){
93            fs.Close();
94          }
95        }
96      }
97
98      /****************************************************************
99        Reads the default XML properties file.
100     ****************************************************************/
101     public void Read(){
102       this.Read(_filename);
103     }
104
105     /***********************************************************
106       Sets a property with given key and value
107     ***********************************************************/
108     public void SetProperty(String key, String value){
109       _properties[key] = value; // overrites old value if it already exists
110     }
111
112     /**********************************************************
113       Gets the value of the specified property key. Will throw an exception
114       if the property does not exist.
115     **********************************************************/
116     public String GetProperty(String key){
117       return _properties[key];
118     }
119  } // end class definition
```

Referring to Example 25.2 — the XmlProperties class is meant to serve as a base class as its two constructors are protected. This class contains the Dictionary<String, String> object which holds the properties as key/value pairs. However, as stated earlier, a generic dictionary object cannot be serialized with the XmlSerializer. To circumvent this limitation I convert the dictionary entries into a list of PropertyEntry objects (List<PropertyEntry>) before serializing them to a file with the XmlSerializer. To trace this conversion see the Store(String filename) method which starts on line 33.

The XmlProperties class has two overloaded Read() methods and two overloaded Store() methods. It also provides a SetProperty() method and a GetProperty() method.

25.3 MyProperties.cs

© 2008 Rick Miller — All Rights Reserved

```
1    using System;
2
3    public class MyProperties : XMLProperties {
4
5       // private fields
6       private static MyProperties _props = null;
7
8       //private constructors
9       private MyProperties() { }
10
11      private MyProperties(String filename):base(filename){ }
12
13      // default GetInstance() method
14      public static MyProperties GetInstance(){
15        return MyProperties.GetInstance(XMLProperties.DEFAULT_PROPERTIES_FILENAME);
16      }
17
18      // GetInstance() method
19      public static MyProperties GetInstance(String filename){
20        if((filename == null) || (filename == String.Empty)){
21          if(_props == null){
22            _props = new MyProperties();
23          }
24        } else {
25          if(_props == null){
26            _props = new MyProperties(filename);
27          }
28        }
29        return _props;
30      }
31   } // end class definition
```

Referring to Example 25.3 — the MyProperties class extends the XMLProperties class and implements the singleton design pattern. Note that both its constructors are private and it contains two overloaded GetInstance() methods. Example 25.4 shows the MyProperties class in action.

25.4 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        MyProperties props = MyProperties.GetInstance();
6        props.SetProperty("Rick", "Pine Forest Senior High");
7        props.SetProperty("Steve", "Pine Forest Senior High");
8        props.SetProperty("Jake", "Pine Forest Senior High");
9        props.SetProperty("Laura", "Pine Forest Senior High");
10       props.SetProperty("Bob", "Pine Forest Senior High");
11       props.Store();
12       props.Read();
13       Console.WriteLine("----------------------------------");
14       Console.WriteLine(props.GetProperty("Rick"));
15       Console.WriteLine(props.GetProperty("Steve"));
16       Console.WriteLine(props.GetProperty("Laura"));
17       Console.WriteLine(props.GetProperty("Jake"));
18     }
19   }
```

Referring to Example 25.4 — a MyProperties reference named props is declared on line 5 and initialized with a call to the MyProperties.GetInstance() method. Next, several properties are set, stored, read, and finally written to the console. The results of running this program are shown in Figure 25-1.The contents of the properties.xml file is given in Example 25.5.

```
Projects
C:\Documents and Settings\Rick\Desktop\Projects\Chapter25\Singleton>mainapp
Rick, Pine Forest Senior High
Steve, Pine Forest Senior High
Jake, Pine Forest Senior High
Laura, Pine Forest Senior High
Bob, Pine Forest Senior High
----------------------------------
Pine Forest Senior High
Pine Forest Senior High
Pine Forest Senior High
Pine Forest Senior High

C:\Documents and Settings\Rick\Desktop\Projects\Chapter25\Singleton>_
```

Figure 25-1: Results of Running Example 25.4

25.5 properties.xml

 © 2008 Rick Miller — All Rights Reserved

```
1    <?xml version="1.0" encoding="utf-8"?>
2    <ArrayOfPropertyEntry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3                          xmlns:xsd="http://www.w3.org/2001/XMLSchema">
4      <PropertyEntry>
5        <PropertyName>Rick</PropertyName>
6        <Value>Pine Forest Senior High</Value>
7      </PropertyEntry>
8      <PropertyEntry>
9        <PropertyName>Steve</PropertyName>
10       <Value>Pine Forest Senior High</Value>
11     </PropertyEntry>
12     <PropertyEntry>
13       <PropertyName>Jake</PropertyName>
14       <Value>Pine Forest Senior High</Value>
15     </PropertyEntry>
16     <PropertyEntry>
17       <PropertyName>Laura</PropertyName>
18       <Value>Pine Forest Senior High</Value>
19     </PropertyEntry>
20     <PropertyEntry>
21       <PropertyName>Bob</PropertyName>
22       <Value>Pine Forest Senior High</Value>
23     </PropertyEntry>
24   </ArrayOfPropertyEntry>
```

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 a class 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*. For more information about these factory patterns please consult the excellent references at the end of this chapter. In this chapter I want to discuss the *dynamic factory* pattern.

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 to dynamically load the class into the .NET virtual machine 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 25.6 gives a simple example of a dynamic factory named InterfaceTypeFactory.

25.6 InterfaceTypeFactory.cs

```
1    using System;
2    using System.Reflection;
3
4    public class InterfaceTypeFactory {
5
6      public static InterfaceType NewObjectByClassName(String classname) {
7        Object o = null;
8        try{
9          Assembly assembly = Assembly.LoadFrom(classname + ".dll");
10         foreach(Type t in assembly.GetTypes()){
11           if(t.Name == classname){
12             o = Activator.CreateInstance(t);
13           }
14         }
15        }catch(Exception e){
16           Console.WriteLine("Problem loading class or creating instance!");
```

 © 2008 Rick Miller — All Rights Reserved

```
17          }
18        return (InterfaceType)o;
19       }
20    } // end InterfaceTypeFactory class definition
```

Referring to Example 25.6 — 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 located in a dynamically linked library (dll) with the name *classname* + ".dll".

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.

25.7 InterfaceType.cs

```
1    public interface InterfaceType {
2      string Message {
3        get;
4      }
5    }
```

25.8 ClassA.cs

```
1    public class ClassA : InterfaceType {
2      private string message = "ClassA's message";
3      public string Message {
4        get { return message; }
5      }
6    }
```

25.9 ClassB.cs

```
1    public class ClassB : InterfaceType {
2      private string message = "ClassB's message";
3      public string Message {
4        get { return message; }
5      }
6    }
```

25.10 ClassC.cs

```
1    public class ClassC : InterfaceType {
2      private string message = "ClassC's message";
3      public string Message {
4        get { return message; }
5      }
6    }
```

25.11 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5       InterfaceType t1 = InterfaceTypeFactory.NewObjectByClassName("ClassA");
6       InterfaceType t2 = InterfaceTypeFactory.NewObjectByClassName("ClassB");
7       InterfaceType t3 = InterfaceTypeFactory.NewObjectByClassName("ClassC");
8
9       Console.WriteLine(t1.Message);
10      Console.WriteLine(t2.Message);
11      Console.WriteLine(t3.Message);
12     }
13   }
```

Ok, to get this example to work, you need to first compile each of the files InterfaceType, ClassA, ClassB, and ClassC into separate dlls. Start with the InterfaceType interface and compile it into a dll with the following command:

```
csc /t:library InterfaceType.cs
```

Next, compile ClassA into a dll with the following command:

```
csc /t:library /r:InterfaceType.dll ClassA.cs
```

Repeat this command to compile ClassB and ClassC into separate dlls as well. At this stage, you should have four dlls named InterfaceType.dll, ClassA.dll, ClassB.dll, and ClassC.dll. Next, compile the InterfaceTypeFactory and MainApp classes together to create the main application with the following command:

```
csc /r:InterfaceType.dll InterfaceTypeFactory.cs MainApp.cs
```

Figure 25-2 shows the results of running this program.

© 2008 Rick Miller — All Rights Reserved

```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter25\DynamicFactory>mainapp
ClassA's message
ClassB's message
ClassC's message

C:\Documents and Settings\Rick\Desktop\Projects\Chapter25\DynamicFactory>_
```

Figure 25-2: 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 application directory 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.

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 with the System.Activator.CreateInstance() method. One of the primary advantages of the dynamic factory pattern is that certain enhancements can be deployed 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 25-3.

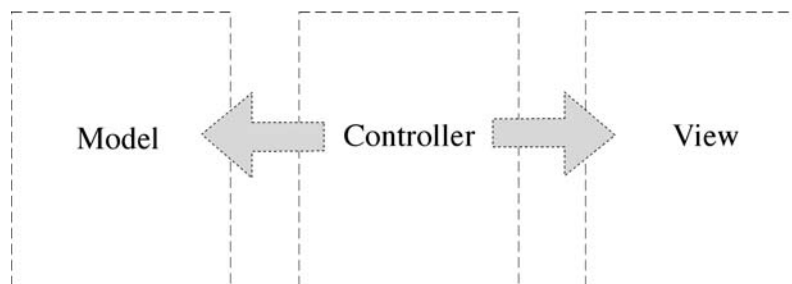


Figure 25-3: Model-View-Controller Pattern

© 2008 Rick Miller — All Rights Reserved

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. For example, you'll need to pass in a reference to the controller object, especially if it's the controller that contains event handler methods you want to assign to buttons or other GUI components.

Examples 25.12 through 25.14 gives the code for a simple implementation of the MVC pattern. I call this application Inspirational Messages. When the application runs it presents a simple user interface consisting of a form and a button. These components are contained in a class named View. When the button is clicked the Controller object handles the Click event and calls a method on the Model object to get the next message, passing it on to 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 example also serves as the application. The results of running this program are shown in Figure 25-4.

25.12 Model.cs

```
1    using System;
2
3    public class Model {
4
5       private int i = 0;
6
7       private String[] messages = { "Eat right, get plenty of rest, and exercise daily.",
8                                     "Make love not war.",
9                                     "Carpe Diem!",
10                                    "Eat your vegatables.",
11                                    "Brush and floss your teeth three times daily.",
12                                    "A penny saved is a penny earned.",
13                                    "What you do today prepares you for tomorrow.",
14                                    "All work and no play makes Jack a dull boy.", };
15
16
17
18      public String GetMessage(){
19        if(i++ == (messages.Length-1)) i = 0;
20        return messages[i];
21       }
22
23   } // end Model class
```

25.13 View.cs

```
1    using System;
2    using System.Drawing;
3    using System.Windows.Forms;
4
5    public class View : Form {
6
7
8       private Button button = null;
9      private Label label = null;
10     private TableLayoutPanel panel = null;
11
12      public View(Controller c){
13        button = new Button();
14       button.Text = "Next Message";
15        button.Click += c.ClickHandler;
16       button.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
17       label = new Label();
18       label.Text = "";
19       label.Font = new Font(label.Font, FontStyle.Bold);
20       label.Height = 50;
21       label.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
22       panel = new TableLayoutPanel();
23       panel.RowCount = 2;
24       panel.ColumnCount = 1;
25       panel.Dock = DockStyle.Top;
26       panel.Height = 100;
27       panel.Controls.Add(label);
28       panel.Controls.Add(button);
29       this.Controls.Add(panel);
30       this.Text = "Inspirational Messages";
31       this.Width = 400;
32       this.Height = 125;
33       this.Visible = true;
34      }
35
36      public void SetMessage(String message){
```

 © 2008 Rick Miller — All Rights Reserved

```
37      label.Text = message;
38      this.Update();
39     }
40
41   } // end View clas definition
```

25.14 Controller.cs

```
1    using System;
2    using System.Windows.Forms;
3
4    public class Controller {
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        Application.Run(its_view);
12      }
13
14      public void ClickHandler(Object sender, EventArgs e){
15           its_view.SetMessage(its_model.GetMessage());
16         }
17
18     public static void Main(){
19        new Controller();
20     }
21
22   } // end Controller class definition
```

Figure 25-4: Results of Running Example 25.11 and Clicking the “Next Message” Button Several Times

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 Control/EventHandler delegate relationship as you saw in the previous section. Components such as Buttons can add any number of EventHandler delegate methods to their Click event. When a button is clicked the assigned event handler methods are called, passing to them a reference to the object that was clicked and an instance of EventArgs. If only one EventHandler delegate method exists to handle all button clicks then it’s the responsibility of the event handler method to determine the source component and perform the necessary actions. In large application this can lead to a large method.

Another approach to implementing the command pattern in C# that you see frequent examples of is to extend components like Button and make them commands. I don’t like this approach because it violates Coad’s criteria. (*i.e., Is a command really a button? I discussed Coad’s criteria in Chapter 24*.)

© 2008 Rick Miller — All Rights Reserved

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 25.15.

25.15 BaseCommand.cs

```
1    using System;
2    using Com.PulpFreePress.Common;
3
4    namespace Com.PulpFreePress.Commands {
5      public abstract class BaseCommand {
6         protected static IModel its_model = null;
7         protected static IView  its_view = null;
8
9         public IModel Model {
10        set {
11            if(its_model == null){
12           its_model = value;
13            }
14          }
15        }
16
17        public IView View {
18         set {
19            if(its_view == null){
20            its_view = value;
21            }
22          }
23        }
24
25        public abstract void Execute(); // must be implemented in derived classes
26
27     } // end BaseCommand class definition
28   } // end namespace
```

Referring to Example 25-15 — 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 properties Model and View 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 event handler method as is shown in Example 25.16.

25.16 Controller.cs

```
1    using System;
2    using System.Windows.Forms;
3    using Com.PulpFreePress.Common;
4    using Com.PulpFreePress.Exceptions;
5    using Com.PulpFreePress.Commands;
6    using Com.PulpFreePress.Model;
7    using Com.PulpFreePress.View;
8    using Com.PulpFreePress.Utils;
9
10   public class Controller : IController {
11
12      private CommandFactory command_factory = null;
13      private IModel its_model;
14      private IView its_view;
15
16      public Controller(){
17        command_factory = CommandFactory.GetInstance();
18        its_model = new Model();
19        its_view = new Com.PulpFreePress.View.View(this);
20        Application.Run((Form)its_view);
21      }
22
23     public void UniversalHandler(Object sender, EventArgs e){
24       try{
25        BaseCommand command = null;
26        if(sender.GetType() == typeof(Button)){
27          command = command_factory.GetCommand(((Button)sender).Name);
28        }else{
29          command = command_factory.GetCommand(((ToolStripMenuItem)sender).Name);
30         }
31        command.Model = its_model;
```

 © 2008 Rick Miller — All Rights Reserved

```
32        command.View = its_view;
33        command.Execute();
34        }catch(CommandNotFoundException cnfe){
35           Console.WriteLine("Command not found!");
36        }
37     }
38
39     public static void Main(){
40        new Controller();
41     } // end Main() method
42   } // end Controller class definition
```

Referring to Example 25.16 — 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 UniversalHandler() method beginning on line 23. The command_factory reference is used to dynamically load and create an instance of a command based on the Name property of a clicked control. (Unfortunately, ToolStripMenuItems are not Controls so I can't treat the two polymorphically, I must use the typeof operator to distinguish between Buttons and ToolStripMenuItems!) Once the command is created, I set its Model and View properties and then call its Execute() method. So long as the command class exists in the Command.dll library things will work as expected.

Example 25.17 gives the code for the CommandFactory class.

25.17 CommandFactory.cs

```
1    using System;
2    using System.Reflection;
3    using Com.PulpFreePress.Commands;
4    using Com.PulpFreePress.Exceptions;
5
6    namespace Com.PulpFreePress.Utils {
7      public class CommandFactory {
8
9      private static CommandFactory command_factory_instance = null;
10     private static CommandProperties command_properties = null;
11
12     static CommandFactory() {
13        command_properties = CommandProperties.GetInstance();
14      }
15
16
17     private CommandFactory(){}
18
19     public static CommandFactory GetInstance(){
20       if(command_factory_instance == null){
21         command_factory_instance = new CommandFactory();
22       }
23       return command_factory_instance;
24     }
25
26     /*****************************************************************
27       Thorws CommandNotFoundException if command does not exist or
28       command_string equals null.
29     *************************************************************/
30     public BaseCommand GetCommand(String command_string){
31       BaseCommand command = null;
32       if(command_string == null){
33          throw new CommandNotFoundException( command_string + " command class not found!");
34        } else{
35        try {
36             Assembly assembly = Assembly.LoadFrom("Commands.dll"); // expect to find commands in Commands.dll
37              String command_type_name = command_properties.GetProperty(command_string);
38           foreach(Type t in assembly.GetTypes()){
39             if(t.Name == command_type_name){
40               command = (BaseCommand) Activator.CreateInstance(t);
41             }
42           }
43             }catch(Exception ex){
44               Console.WriteLine(ex);
45               throw new CommandNotFoundException(ex.ToString(), ex);
46              }
47         } // end else
48         return command;
49      } // end etCommand() method
50
51     } // end CommandFactory class definition
52   } // end namespace
```


© 2008 Rick Miller — All Rights Reserved

Referring to Example 25.17 — The CommandFactory implements the singleton pattern. Most of the work of this class is done in the GetCommand() method. Here the command_string argument is used to look up the 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. This class also utilizes the services of the CommandProperties class which is presented in Example 25.18.

25.18 CommandProperties.cs

```
1     using System;
2
3     namespace Com.PulpFreePress.Utils {
4       public class CommandProperties : XMLProperties {
5
6       // class constants - default key strings
7        public const String PROPERTIES_FILE              = "PROPERTIES_FILE";
8        public const String NEWHOURLYEMPLOYEE_COMMAND    = "NewHourlyEmployee";
9        public const String NEWSALARIEDEMPLOYEE_COMMAND  = "NewSalariedEmployee";
10       public const String EXIT_COMMAND                 = "Exit";
11       public const String LIST_COMMAND                 = "List";
12       public const String SORT_COMMAND                 = "Sort";
13       public const String SAVE_COMMAND                 = "Save";
14       public const String EDITEMPLOYEE_COMMAND         = "EditEmployee";
15       public const String DELETEEMPLOYEE_COMMAND       = "DeleteEmployee";
16       public const String LOAD_COMMAND                 = "Load";
17       public const String CLEAR_COMMAND                = "Clear";
18       public const String SUBMIT_COMMAND               = "Submit";
19
20
21       // class constants - default value strings
22       private const String PROPERTIES_FILE_VALUE
23                                                       = "CommandProperties.XML";
24       private const String NEWHOURLYEMPLOYEE_COMMAND_CLASSNAME       = "NewHourlyEmployeeCommand";
25       private const String NEWSALARIEDEMPLOYEE_COMMAND_CLASSNAME     = "NewSalariedEmployeeCommand";
26       private const String EXIT_COMMAND_CLASSNAME                    = "ApplicationExitCommand";
27       private const String LIST_COMMAND_CLASSNAME                    = "ListEmployeesCommand";
28       private const String SORT_COMMAND_CLASSNAME                    = "SortEmployeesCommand";
29       private const String SAVE_COMMAND_CLASSNAME                    = "SaveEmployeesCommand";
30       private const String EDITEMPLOYEE_COMMAND_CLASSNAME            = "EditEmployeeCommand";
31       private const String DELETEEMPLOYEE_COMMAND_CLASSNAME          = "DeleteEmployeeCommand";
32       private const String LOAD_COMMAND_CLASSNAME                    = "LoadEmployeesCommand";
33       private const String CLEAR_COMMAND_CLASSNAME                   = "ClearInputFieldsCommand";
34       private const String SUBMIT_COMMAND_CLASSNAME                  = "SubmitCommand";
35
36
37       // private fields
38       private static CommandProperties _props = null;
39
40       //private constructor
41       private CommandProperties():this(PROPERTIES_FILE_VALUE) { }
42
43       private CommandProperties(String filename):base(filename){
44         SetProperty(PROPERTIES_FILE, PROPERTIES_FILE_VALUE);
45         SetProperty(NEWHOURLYEMPLOYEE_COMMAND, NEWHOURLYEMPLOYEE_COMMAND_CLASSNAME);
46         SetProperty(NEWSALARIEDEMPLOYEE_COMMAND, NEWSALARIEDEMPLOYEE_COMMAND_CLASSNAME);
47         SetProperty(EXIT_COMMAND, EXIT_COMMAND_CLASSNAME);
48         SetProperty(LIST_COMMAND, LIST_COMMAND_CLASSNAME);
49         SetProperty(SORT_COMMAND, SORT_COMMAND_CLASSNAME);
50         SetProperty(SAVE_COMMAND, SAVE_COMMAND_CLASSNAME);
51         SetProperty(EDITEMPLOYEE_COMMAND, EDITEMPLOYEE_COMMAND_CLASSNAME);
52         SetProperty(DELETEEMPLOYEE_COMMAND, DELETEEMPLOYEE_COMMAND_CLASSNAME);
53         SetProperty(LOAD_COMMAND, LOAD_COMMAND_CLASSNAME);
54         SetProperty(CLEAR_COMMAND, CLEAR_COMMAND_CLASSNAME);
55         SetProperty(SUBMIT_COMMAND, SUBMIT_COMMAND_CLASSNAME);
56         base.Store();
57       }
58
59       // default GetInstance() method
60       public static CommandProperties GetInstance(){
61         return CommandProperties.GetInstance(PROPERTIES_FILE_VALUE);
62       }
63
64       // GetInstance() method
65       public static CommandProperties GetInstance(String filename){
66         if((filename == null) || (filename == String.Empty)){
67           if(_props == null){
68             _props = new CommandProperties();
69           }
70         } else {
71           if(_props == null){
72             _props = new CommandProperties(filename);
```


© 2008 Rick Miller — All Rights Reserved

```
73          }
74        }
75        return _props;
76      }
77    } // end class definition
78  } // end namespace
```

Referring to Example 25.18 — the CommandProperties class extends the XMLProperties class presented earlier in Example 25.2. The CommandProperties class defines two types of string constants: 1) *command names*, and 2) their *corresponding class names*. When an instance of the CommandProperties class is created, these string constants are then used as key/value pairs to set the dictionary values of the XMLProperties class via the SetProperty() method. The Store() method is then called to persist the properties in an XML file that can later be manually edited if more commands are added. (Or, command properties can be added via the CommandProperties object programmatically.)

Command strings are mapped to their respective command classes in the CommandProperties.XML file which is shown in Example 25.19.

25.19 CommandProperties.XML file contents

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <ArrayOfPropertyEntry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3                         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
4
5     <PropertyEntry>
6       <PropertyName>PROPERTIES_FILE</PropertyName>
7       <Value>CommandProperties.XML</Value>
8     </PropertyEntry>
9     <PropertyEntry>
10      <PropertyName>NewHourlyEmployee</PropertyName>
11      <Value>NewHourlyEmployeeCommand</Value>
12    </PropertyEntry>
13    <PropertyEntry>
14      <PropertyName>NewSalariedEmployee</PropertyName>
15      <Value>NewSalariedEmployeeCommand</Value>
16    </PropertyEntry>
17    <PropertyEntry>
18      <PropertyName>Exit</PropertyName>
19      <Value>ApplicationExitCommand</Value>
20    </PropertyEntry>
21    <PropertyEntry>
22      <PropertyName>List</PropertyName>
23      <Value>ListEmployeesCommand</Value>
24    </PropertyEntry>
25    <PropertyEntry>
26      <PropertyName>Sort</PropertyName>
27      <Value>SortEmployeesCommand</Value>
28    </PropertyEntry>
29    <PropertyEntry>
30      <PropertyName>Save</PropertyName>
31      <Value>SaveEmployeesCommand</Value>
32    </PropertyEntry>
33    <PropertyEntry>
34      <PropertyName>EditEmployee</PropertyName>
35      <Value>EditEmployeeCommand</Value>
36    </PropertyEntry>
37    <PropertyEntry>
38      <PropertyName>DeleteEmployee</PropertyName>
39      <Value>DeleteEmployeeCommand</Value>
40    </PropertyEntry>
41    <PropertyEntry>
42      <PropertyName>Load</PropertyName>
43      <Value>LoadEmployeesCommand</Value>
44    </PropertyEntry>
45    <PropertyEntry>
46      <PropertyName>Clear</PropertyName>
47      <Value>ClearInputFieldsCommand</Value>
48    </PropertyEntry>
49    <PropertyEntry>
50      <PropertyName>Submit</PropertyName>
51      <Value>SubmitCommand</Value>
52    </PropertyEntry>
53  </ArrayOfPropertyEntry>
```


© 2008 Rick Miller — All Rights Reserved

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 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.

Complete Code Listing

This section gives the complete code listing for the comprehensive pattern example by namespace. This code resides in the folder named EmployeeMVC in the Chapter 25 projects folder. Figure 25-5 shows the project folder directory structure.

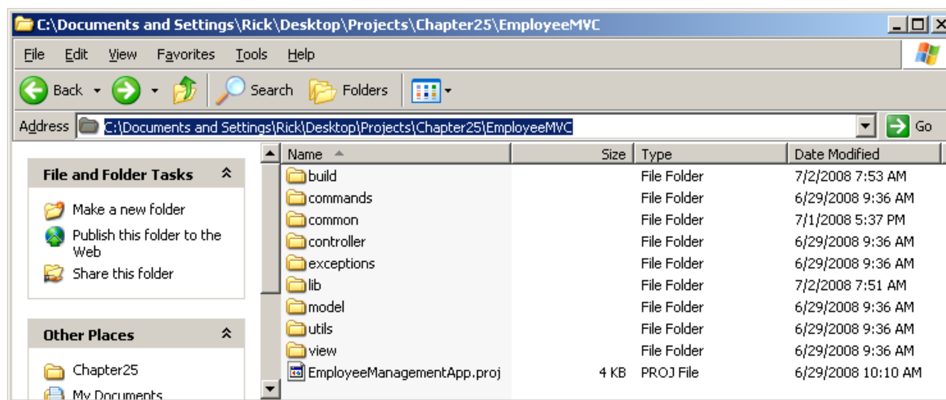


Figure 25-5: EmployeeMVC Project Directory Structure

Com.PulpFreePress.Exceptions

25.20 CommandNotFoundException.cs

```
1     using System;
2
3     namespace Com.PulpFreePress.Exceptions {
4       public class CommandNotFoundException : Exception {
5         public CommandNotFoundException(String message, Exception ex):base(message, ex){}
6
7         public CommandNotFoundException(String message):base(message){}
8
9         public CommandNotFoundException():this("Command not found exception"){ }
10      } // end CommandNotFoundException class definition
11    } // end namespace
```

Com.PulpFreePress.Common

25.21 IEmployee.cs

```
1     using System;
2
3     namespace Com.PulpFreePress.Common {
```

 © 2008 Rick Miller — All Rights Reserved

```
4
5      public interface IEmployee : IComparable<IEmployee> {
6         int Age { get; }
7         String FullName { get; }
8         String FullNameAndAge { get; }
9         String FirstName { get; set; }
10        String MiddleName { get; set; }
11        String LastName { get; set; }
12        String EmployeeNumber { get; set; }
13        DateTime Birthday { get; set; }
14        Sex Gender { get; set; }
15        PayInfo PayInfo { set; get; }
16        double Pay { get; }
17     }
18   } //namespace
```

25.22 Person.cs

```
1    using System;
2
3    namespace Com.PulpFreePress.Common {
4      [Serializable]
5      public class Person : IComparable<Person> {
6
7        // private instance fields
8        private String   _firstName;
9        private String   _middleName;
10       private String   _lastName;
11       private Sex      _gender;
12       private DateTime _birthday;
13
14
15       //private default constructor
16       public Person(){}
17
18       public Person(String firstName, String middleName, String lastName,
19                     Sex gender, DateTime birthday){
20          FirstName = firstName;
21          MiddleName = middleName;
22          LastName = lastName;
23          Gender = gender;
24          BirthDay = birthday;
25       }
26
27       // public properties
28       public String FirstName {
29         get { return _firstName; }
30         set { _firstName = value; }
31       }
32
33       public String MiddleName {
34         get { return _middleName; }
35         set { _middleName = value; }
36       }
37
38       public String LastName {
39         get { return _lastName; }
40         set { _lastName = value; }
41       }
42
43       public Sex Gender {
44         get { return _gender; }
45         set { _gender = value; }
46       }
47
48       public DateTime BirthDay {
49         get { return _birthday; }
50         set { _birthday = value; }
51       }
52
53      public int Age {
54        get {
55         int years = DateTime.Now.Year - _birthday.Year;
56         int adjustment = 0;
57           if(DateTime.Now.Month < _birthday.Month){
58             adjustment = 1;
59           }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
60                     adjustment = 1;
61                 }
62         return years - adjustment;
63        }
```

 © 2008 Rick Miller — All Rights Reserved

```
64        }
65
66        public String FullName {
67          get { return FirstName + " " + MiddleName + " " + LastName; }
68        }
69
70        public String FullNameAndAge {
71          get { return FullName + " " + Age; }
72        }
73
74        public override String ToString(){
75          return FullName + ", " + Gender + ", " + Age;
76        }
77
78        public int CompareTo(Person other){
79          return this.FullName.CompareTo(other.FullName);
80        }
81
82      } // end Person class
83    } // namespace
```

25.23 Employee.cs

```
84    using System;
85
86    namespace Com.PulpFreePress.Common {
87      [Serializable]
88      public abstract class Employee : IEmployee {
89         private Person _person = null;
90         private String  _employee_number = null;
91         private PayInfo _payInfo = null;
92
93         protected Employee(){
94           _person = new Person();
95         }
96
97         protected Employee(String f_name, String m_name, String l_name, Sex gender, DateTime birthday,
98                            String employee_number){
99            _person = new Person(f_name, m_name, l_name, gender, birthday );
100           _employee_number = employee_number;
101         } // end constructor
102
103        public int Age {
104          get{ return _person.Age; }
105         }
106        public String FullName {
107           get { return _person.FullName; }
108        }
109        public String FullNameAndAge {
110          get { return _person.FullNameAndAge; }
111        }
112        public String FirstName {
113          get { return _person.FirstName; }
114          set { _person.FirstName = value; }
115       }
116        public String MiddleName {
117          get { return _person.MiddleName; }
118          set { _person.MiddleName = value; }
119        }
120        public String LastName {
121          get { return _person.LastName; }
122          set { _person.LastName = value; }
123        }
124        public Sex Gender {
125          get { return _person.Gender; }
126          set { _person.Gender = value; }
127        }
128        public String EmployeeNumber {
129           get { return _employee_number; }
130           set { _employee_number = value; }
131        }
132        public DateTime Birthday {
133          get { return _person.BirthDay; }
134          set { _person.BirthDay = value; }
135        }
136
137        public PayInfo PayInfo {
138          get { return _payInfo; }
139          set { _payInfo = value; }
140        }
141
```

 © 2008 Rick Miller — All Rights Reserved

```
142       // defer implementation of this property
143       public abstract double Pay { get; }
144
145       public override String ToString(){
146         return (_person.ToString() + " " + EmployeeNumber + " ");
147       }
148
149       public int CompareTo(IEmployee other){
150         return this.ToString().CompareTo(other.ToString());
151       }
152    }  // end Employee class definition
153  } // namespace
```

25.24 HourlyEmployee.cs

```
1    using System;
2
3    namespace Com.PulpFreePress.Common {
4      [Serializable]
5      public class HourlyEmployee : Employee {
6
7        public HourlyEmployee():base() { }
8
9        public HourlyEmployee(String f_name, String m_name, String l_name, Sex gender, DateTime birthday,
10                             String employee_number)
11             :base(f_name, m_name, l_name, gender, birthday,  employee_number){ }
12
13        public override double Pay {
14          get { return base.PayInfo.HoursWorked * base.PayInfo.HourlyRate; }
15        }
16
17        public override String ToString() {
18          return (base.ToString() + " " + Pay.ToString("C3"));
19
20        }
21     } // end HourlyEmployee class definition
22   } // namespace
```

25.25 SalariedEmployee.cs

```
1    using System;
2
3    namespace Com.PulpFreePress.Common {
4      [Serializable]
5      public class SalariedEmployee : Employee {
6
7        public SalariedEmployee():base() { }
8
9        public SalariedEmployee(String f_name, String m_name, String l_name, Sex gender, DateTime birthday,
10                               String employee_number)
11         :base(f_name, m_name, l_name, gender, birthday, employee_number){ }
12
13        public override double Pay {
14          get { return ((base.PayInfo.Salary/12.0)/2.0); }
15       }
16
17        public override String ToString() {
18          return (base.ToString() + " " + Pay.ToString("C3"));
19        }
20     } // end SalariedEmployee class definition
21   } // namespace
```

25.26 PayInfo.cs

```
1    using System;
2
3    namespace Com.PulpFreePress.Common {
4       [Serializable]
5      public class PayInfo {
6        // fields
7        private double _salary = 0;
8        private double _hours_worked = 0;
9        private double _hourly_rate = 0;
10
11       // properties
12       public double Salary {
13          get { return _salary; }
14          set { _salary = value; }
15       }
16
```


© 2008 Rick Miller — All Rights Reserved

```
17       public double HoursWorked {
18         get { return _hours_worked; }
19         set { _hours_worked = value; }
20       }
21
22       public double HourlyRate {
23         get { return _hourly_rate; }
24         set { _hourly_rate = value; }
25       }
26
27       // constructors
28       public PayInfo(){ }
29
30       public PayInfo(double salary){
31          _salary = salary;
32       }
33       public PayInfo(double hours_worked, double hourly_rate){
34          _hours_worked = hours_worked;
35          _hourly_rate = hourly_rate;
36       }
37     } // end PayInfo class definition
38   } // namespace
```

25.27 IModel.cs

```
1    using System;
2    using System.IO;
3
4    namespace Com.PulpFreePress.Common {
5      public interface IModel {
6        void AddEmployee(IEmployee employee);
7
8        void EditEmployee(IEmployee employee, int index);
9
10       String[] GetAllEmployeesInfo();
11
12       IEmployee GetEmployeeByEmployeeNumber(String employee_number);
13
14       IEmployee GetEmployeeByIndex(int index);
15
16       void SortEmployees();
17
18       void DeleteEmployeeByIndex(int index);
19
20       void SaveEmployeesToFile(String filename);
21
22       void LoadEmployeesFromFile(String filename);
23
24     } // end IModel interface definition
25   } // end namespace
```

25.28 IView.cs

```
1    using System;
2    using System.IO;
3
4    namespace Com.PulpFreePress.Common {
5      public interface IView {
6         IEmployee EditingEmployee { get; set; }
7         String FirstName { get; set; }
8         String MiddleName { get; set; }
9         String LastName { get; set; }
10        String EmployeeNumber { get; set; }
11        String Salary { get; set; }
12        String HoursWorked { get; set; }
13        String HourlyRate { get; set; }
14        DateTime Birthday { get; set; }
15        Sex Gender { get; set; }
16        IEmployee GetNewSalariedEmployee();
17        IEmployee GetNewHourlyEmployee();
18        IEmployee GetEditedEmployee();
19        void PopulateEditFields(HourlyEmployee employee);
20        void PopulateEditFields(SalariedEmployee employee);
21        ViewMode Mode { get; set; }
22        void DisplayEmployeeInfo(String[] employees_info);
23        String GetSaveFile();
24        String GetLoadFile();
25        void ClearInputFields();
26        void EnableSubmitButton(bool state);
27        void EnableSalaryFields(bool state);
28        void EnableHourlyFields(bool state);
```


© 2008 Rick Miller — All Rights Reserved

```
29        void SetWindowTitleBasedOnMode();
30        int SelectedLineNumber();
31      } // end IView interface definition
32    } // end namespace
```

25.29 SexEnum.cs

```
1     using System;
2
3     namespace Com.PulpFreePress.Common {
4       //enumeration
5      [Serializable]
6       public enum Sex {MALE, FEMALE}
7     } // end namespace
```

25.30 ViewModeEnum.cs

```
1     namespace Com.PulpFreePress.Common {
2       public enum ViewMode { RESTING, SALARIED, HOURLY, EDIT }
3     }
```

25.31 IController.cs

```
1     using System;
2
3     namespace Com.PulpFreePress.Common {
4       public interface IController {
5         void UniversalHandler(Object sender, EventArgs e);
6       }
7     } // end namespace
```

Com.PulpFreePress.Utils

```
1     PropertyEntry.cs
2     namespace Com.PulpFreePress.Utils {
3       public struct PropertyEntry {
4         public string PropertyName;
5         public string Value;
6       }
7     } // end namespace definition
```

25.32 XMLProperties.cs

```
1     using System;
2     using System.Collections.Generic;
3     using System.IO;
4     using System.Xml;
5     using System.Xml.Serialization;
6
7     namespace Com.PulpFreePress.Utils {
8       public class XMLProperties {
9          // private fields
10         private Dictionary<String, String> _properties = null;
11         private String _filename = null;
12
13         // constants
14         protected const String DEFAULT_PROPERTIES_FILENAME = "properties.xml";
15
16         // protected constructors
17         protected XMLProperties():this(DEFAULT_PROPERTIES_FILENAME){ }
18
19         protected XMLProperties(String filename){
20           if((filename == null) || (filename == String.Empty)){
21             _filename = DEFAULT_PROPERTIES_FILENAME;
22           }else {
23             _filename = filename;
24           }
25           _properties = new Dictionary<String, String>();
26
27         }
28
29         /**************************************************************
30            Converts _properties dictionary into List<PropertyEntry> object
31            and serializes it to an xml file.
32         ************************************************************/
33         public void Store(String filename){
34           TextWriter writer = null;
35           try {
36             writer = new StreamWriter(filename);
37             List<PropertyEntry> entry_list = new List<PropertyEntry>();
```


© 2008 Rick Miller — All Rights Reserved

```
38            foreach(KeyValuePair<String, String> entry in _properties){
39              PropertyEntry pe;
40              pe.PropertyName = entry.Key;
41              pe.Value = entry.Value;
42              entry_list.Add(pe);
43            }
44            //remove
45            foreach(PropertyEntry entry in entry_list){
46              Console.WriteLine(entry.PropertyName + ", " + entry.Value);
47            }
48
49            XmlSerializer serializer = new XmlSerializer(typeof(List<PropertyEntry>));
50            serializer.Serialize(writer, entry_list);
51          } catch(IOException ioe){
52            Console.WriteLine(ioe);
53          } catch(Exception ex){
54            Console.WriteLine(ex);
55          } finally {
56            if(writer != null) writer.Close();
57          }
58          _filename = filename;
59        }
60
61        /**************************************************************
62           Stores property entries to default file
63        **************************************************************/
64        public void Store(){
65          this.Store(_filename);
66        }
67
68        /**************************************************************
69           Reads the XML properties file and populates the _properties
70           dictionary. Throws an IOException if the specified filename does
71           not exist.
72        ************************************************************/
73        public void Read(String filename ){
74
75          if(!File.Exists(filename)){
76            throw new IOException("Requested file does not exist!");
77          }
78          FileStream fs = null;
79          try {
80            fs = new FileStream(filename, FileMode.Open);
81            XmlSerializer serializer = new XmlSerializer(typeof(List<PropertyEntry>));
82            List<PropertyEntry> entry_list = (List<PropertyEntry>)serializer.Deserialize(fs);
83            foreach(PropertyEntry entry in entry_list){
84              _properties[entry.PropertyName] = entry.Value;
85            }
86
87          }catch(IOException ioe){
88            Console.WriteLine(ioe);
89          }catch(Exception ex){
90            Console.WriteLine(ex);
91          }finally{
92            if(fs != null){
93              fs.Close();
94            }
95          }
96        }
97
98        /****************************************************************
99           Reads the default XML properties file.
100       ****************************************************************/
101       public void Read(){
102         this.Read(_filename);
103       }
104
105       /***********************************************************
106          Sets a property with given key and value
107       ***********************************************************/
108       public void SetProperty(String key, String value){
109         _properties[key] = value; // overrites old value if it already exists
110       }
111
112       /**********************************************************
113          Gets the value of the specified property key. Will throw an exception
114          if the property does not exist.
115       ********************************************************/
116       public String GetProperty(String key){
117         return _properties[key];
118       }
```


© 2008 Rick Miller — All Rights Reserved

```
119     } // end class definition
120    } // end namespace
```

25.33 CommandProperties.cs

```
1     using System;
2
3     namespace Com.PulpFreePress.Utils {
4       public class CommandProperties : XMLProperties {
5
6       // class constants - default key strings
7        public const String PROPERTIES_FILE                = "PROPERTIES_FILE";
8        public const String NEWHOURLYEMPLOYEE_COMMAND      = "NewHourlyEmployee";
9        public const String NEWSALARIEDEMPLOYEE_COMMAND    = "NewSalariedEmployee";
10       public const String EXIT_COMMAND                   = "Exit";
11       public const String LIST_COMMAND                   = "List";
12       public const String SORT_COMMAND                   = "Sort";
13       public const String SAVE_COMMAND                   = "Save";
14       public const String EDITEMPLOYEE_COMMAND           = "EditEmployee";
15       public const String DELETEEMPLOYEE_COMMAND         = "DeleteEmployee";
16       public const String LOAD_COMMAND                   = "Load";
17       public const String CLEAR_COMMAND                  = "Clear";
18       public const String SUBMIT_COMMAND                 = "Submit";
19
20
21       // class constants - default value strings
22       private const String PROPERTIES_FILE_VALUE
23                                                      = "CommandProperties.XML";
24       private const String NEWHOURLYEMPLOYEE_COMMAND_CLASSNAME     = "NewHourlyEmployeeCommand";
25       private const String NEWSALARIEDEMPLOYEE_COMMAND_CLASSNAME   = "NewSalariedEmployeeCommand";
26       private const String EXIT_COMMAND_CLASSNAME                  = "ApplicationExitCommand";
27       private const String LIST_COMMAND_CLASSNAME                  = "ListEmployeesCommand";
28       private const String SORT_COMMAND_CLASSNAME                  = "SortEmployeesCommand";
29       private const String SAVE_COMMAND_CLASSNAME                  = "SaveEmployeesCommand";
30       private const String EDITEMPLOYEE_COMMAND_CLASSNAME          = "EditEmployeeCommand";
31       private const String DELETEEMPLOYEE_COMMAND_CLASSNAME        = "DeleteEmployeeCommand";
32       private const String LOAD_COMMAND_CLASSNAME                  = "LoadEmployeesCommand";
33       private const String CLEAR_COMMAND_CLASSNAME                 = "ClearInputFieldsCommand";
34       private const String SUBMIT_COMMAND_CLASSNAME                = "SubmitCommand";
35
36       // private fields
37       private static CommandProperties _props = null;
38
39       //private constructor
40       private CommandProperties():this(PROPERTIES_FILE_VALUE) { }
41
42       private CommandProperties(String filename):base(filename){
43         SetProperty(PROPERTIES_FILE, PROPERTIES_FILE_VALUE);
44         SetProperty(NEWHOURLYEMPLOYEE_COMMAND, NEWHOURLYEMPLOYEE_COMMAND_CLASSNAME);
45         SetProperty(NEWSALARIEDEMPLOYEE_COMMAND, NEWSALARIEDEMPLOYEE_COMMAND_CLASSNAME);
46         SetProperty(EXIT_COMMAND, EXIT_COMMAND_CLASSNAME);
47         SetProperty(LIST_COMMAND, LIST_COMMAND_CLASSNAME);
48         SetProperty(SORT_COMMAND, SORT_COMMAND_CLASSNAME);
49         SetProperty(SAVE_COMMAND, SAVE_COMMAND_CLASSNAME);
50         SetProperty(EDITEMPLOYEE_COMMAND, EDITEMPLOYEE_COMMAND_CLASSNAME);
51         SetProperty(DELETEEMPLOYEE_COMMAND, DELETEEMPLOYEE_COMMAND_CLASSNAME);
52         SetProperty(LOAD_COMMAND, LOAD_COMMAND_CLASSNAME);
53         SetProperty(CLEAR_COMMAND, CLEAR_COMMAND_CLASSNAME);
54         SetProperty(SUBMIT_COMMAND, SUBMIT_COMMAND_CLASSNAME);
55         base.Store();
56       }
57
58       // default GetInstance() method
59       public static CommandProperties GetInstance(){
60         return CommandProperties.GetInstance(PROPERTIES_FILE_VALUE);
61       }
62
63       // GetInstance() method
64       public static CommandProperties GetInstance(String filename){
65         if((filename == null) || (filename == String.Empty)){
66           if(_props == null){
67             _props = new CommandProperties();
68           }
69         } else {
70           if(_props == null){
71             _props = new CommandProperties(filename);
72           }
73         }
74         return _props;
75       }
76      } // end class definition
77    } // end namespace
```


© 2008 Rick Miller — All Rights Reserved

Com.PulpFreePress.Commands

25.34 BaseCommand.cs

```
1    using System;
2    using Com.PulpFreePress.Common;
3
4    namespace Com.PulpFreePress.Commands {
5      public abstract class BaseCommand {
6         protected static IModel its_model = null;
7         protected static IView  its_view = null;
8
9         public IModel Model {
10        set {
11            if(its_model == null){
12           its_model = value;
13            }
14          }
15        }
16
17        public IView View {
18         set {
19            if(its_view == null){
20            its_view = value;
21            }
22          }
23        }
24
25        public abstract void Execute(); // must be implemented in derived classes
26
27     } // end BaseCommand class definition
28   } // end namespace
```

25.35 ApplicationExitCommand.cs

```
1    using System;
2    using System.Windows.Forms;
3
4    namespace Com.PulpFreePress.Commands {
5      public class ApplicationExitCommand : BaseCommand {
6        public override void Execute(){
7          Application.Exit();
8        }
9      }
10   } // end namespace
```

25.36 ClearInputFieldsCommand.cs

```
1    using System;
2    using Com.PulpFreePress.Common;
3
4    namespace Com.PulpFreePress.Commands {
5      public class ClearInputFieldsCommand : BaseCommand {
6        public override void Execute(){
7          if(its_view != null){
8            its_view.ClearInputFields();
9            its_view.EnableSubmitButton(false);
10           its_view.EnableHourlyFields(false);
11           its_view.EnableSalaryFields(false);
12           its_view.Mode = ViewMode.RESTING;
13         }
14       } // end Execute() method
15     } // end NewSalariedEmployeeCommand class definition
16   } // end namespace
```

25.37 DeleteEmployeeCommand.cs

```
1    using System;
2
3    namespace Com.PulpFreePress.Commands {
4      public class DeleteEmployeeCommand : BaseCommand {
5
6        public override void Execute(){
7          if((its_model != null) && (its_view != null)){
8             int index = its_view.SelectedLineNumber();
9             Console.WriteLine(index);
10            its_model.DeleteEmployeeByIndex(index);
11            its_view.DisplayEmployeeInfo(its_model.GetAllEmployeesInfo());
```


© 2008 Rick Miller — All Rights Reserved

```
12            its_view.ClearInputFields();
13          }
14        } // end Execute() method
15      } // end DeleteEmployeeCommand class definition
16    } // end namespace
```

25.38 EditEmployeeCommand.cs

```
1     using System;
2     using Com.PulpFreePress.Common;
3
4     namespace Com.PulpFreePress.Commands {
5       public class EditEmployeeCommand : BaseCommand {
6
7         public override void Execute(){
8           if((its_model != null) && (its_view != null)){
9              int index = its_view.SelectedLineNumber();
10             Console.WriteLine(index);
11             IEmployee employee = its_model.GetEmployeeByIndex(index);
12             if(employee != null){
13               its_view.EditingEmployee = employee;
14               its_view.EnableSubmitButton(true);
15               its_view.Mode = ViewMode.EDIT;
16             }
17          }
18        } // end Execute() method
19      } // end EditEmployeeCommand class definition
20    } // end namespace
```

25.39 ListEmployeesCommand.cs

```
1     using System;
2
3     namespace Com.PulpFreePress.Commands {
4       public class ListEmployeesCommand : BaseCommand {
5         public override void Execute(){
6           if((its_model != null) && (its_view != null)){
7              its_model.LoadEmployeesFromFile(null); // will load default data file
8              its_view.DisplayEmployeeInfo(its_model.GetAllEmployeesInfo());
9           }
10        }
11      }
12    } // end namespace
```

25.40 LoadEmployeesCommand.cs

```
1     using System;
2
3     namespace Com.PulpFreePress.Commands {
4       public class LoadEmployeesCommand : BaseCommand {
5
6         public override void Execute(){
7           if((its_model != null) && (its_view != null)){
8              its_model.LoadEmployeesFromFile(its_view.GetLoadFile());
9
10          }
11          its_view.DisplayEmployeeInfo(its_model.GetAllEmployeesInfo());
12        } // end Execute() method
13      } // end DeleteEmployeeCommand class definition
14    } // end namespace
```

25.41 NewHourlyEmployeeCommand.cs

```
1     using System;
2     using Com.PulpFreePress.Common;
3
4     namespace Com.PulpFreePress.Commands {
5       public class NewHourlyEmployeeCommand : BaseCommand {
6         public override void Execute(){
7           if(its_view != null){
8             its_view.EnableHourlyFields(true);
9             its_view.EnableSalaryFields(false);
10            its_view.ClearInputFields();
11            its_view.EnableSubmitButton(true);
12            its_view.Mode = ViewMode.HOURLY;
13            its_view.SetWindowTitleBasedOnMode();
14          }
15        } // end Execute() method
16      } // end NewHourlyEmployeeCommand class definition
17    } // end namespace
```

 © 2008 Rick Miller — All Rights Reserved

25.42 NewSalariedEmployeeCommand.cs

```
1   using System;
2   using Com.PulpFreePress.Common;
3
4   namespace Com.PulpFreePress.Commands {
5     public class NewSalariedEmployeeCommand : BaseCommand {
6       public override void Execute(){
7         if(its_view != null){
8           its_view.EnableHourlyFields(false);
9           its_view.EnableSalaryFields(true);
10          its_view.ClearInputFields();
11          its_view.EnableSubmitButton(true);
12          its_view.Mode = ViewMode.SALARIED;
13          its_view.SetWindowTitleBasedOnMode();
14        }
15      } // end Execute() method
16    } // end NewSalariedEmployeeCommand class definition
17  } // end namespace
```

25.43 SaveEmployeesCommand.cs

```
1   using System;
2
3   namespace Com.PulpFreePress.Commands {
4     public class SaveEmployeesCommand : BaseCommand {
5
6       public override void Execute(){
7         if((its_model != null) && (its_view != null)){
8            its_model.SaveEmployeesToFile(its_view.GetSaveFile());
9
10        }
11        its_view.DisplayEmployeeInfo(its_model.GetAllEmployeesInfo());
12      } // end Execute() method
13    } // end DeleteEmployeeCommand class definition
14  } // end namespace
```

25.44 SortEmployeesCommand.cs

```
1   using System;
2   using Com.PulpFreePress.Common;
3
4   namespace Com.PulpFreePress.Commands {
5     public class SortEmployeesCommand : BaseCommand {
6       public override void Execute(){
7         if(its_model != null){
8            its_model.SortEmployees();
9            its_view.DisplayEmployeeInfo(its_model.GetAllEmployeesInfo());
10        }
11      } // end Execute() method
12    } // end SortEmployeesCommand class definition
13  } // end namespace
```

25.45 SubmitCommand.cs

```
1   using System;
2   using Com.PulpFreePress.Common;
3
4   namespace Com.PulpFreePress.Commands {
5     public class SubmitCommand : BaseCommand {
6       public override void Execute(){
7        if((its_view != null) && (its_model != null)){
8          switch(its_view.Mode){
9            case ViewMode.SALARIED:
10                   its_model.AddEmployee(its_view.GetNewSalariedEmployee());
11                    its_view.DisplayEmployeeInfo(its_model.GetAllEmployeesInfo());
12                    its_view.ClearInputFields();
13                    its_view.EnableSubmitButton(false);
14                    its_view.EnableSalaryFields(false);
15                    its_view.Mode = ViewMode.RESTING;
16                   break;
17           case ViewMode.HOURLY:
18                   its_model.AddEmployee(its_view.GetNewHourlyEmployee());
19                    its_view.DisplayEmployeeInfo(its_model.GetAllEmployeesInfo());
20                    its_view.ClearInputFields();
21                    its_view.EnableSubmitButton(false);
22                    its_view.EnableHourlyFields(false);
23                    its_view.Mode = ViewMode.RESTING;
24                   break;
25           case ViewMode.EDIT:
26                   int index = its_view.SelectedLineNumber();
```


© 2008 Rick Miller — All Rights Reserved

```
27                    its_model.EditEmployee(its_view.GetEditedEmployee(), index);
28                    its_view.DisplayEmployeeInfo(its_model.GetAllEmployeesInfo());
29                    its_view.ClearInputFields();
30                    its_view.EnableSubmitButton(false);
31                    its_view.EnableHourlyFields(false);
32                    its_view.EnableSalaryFields(false);
33                    its_view.Mode = ViewMode.RESTING;
34                    break;
35          }
36        }
37      } // end Execute() method
38
39    } // end NewSalariedEmployeeCommand class definition
40  } // end namespace
```

Com.PulpFreePress.Model

25.46 Model.cs

```
1     using System;
2     using System.Collections.Generic;
3     using System.Runtime.Serialization;
4     using System.Runtime.Serialization.Formatters.Binary;
5     using System.IO;
6     using Com.PulpFreePress.Common;
7
8     namespace Com.PulpFreePress.Model {
9       public class Model : IModel {
10
11        private List<IEmployee> _employee_list = null;
12        private IEmployeeFactory _employee_factory = null;
13
14        public Model(){
15           _employee_list = new List<IEmployee>();
16           _employee_factory = new EmployeeFactory();
17        }
18
19        public void AddEmployee(IEmployee employee){
20          _employee_list.Add(employee);
21        }
22
23        public void EditEmployee(IEmployee employee, int index){
24          _employee_list[index] = employee;
25        }
26
27
28
29        public String[] GetAllEmployeesInfo(){
30          String[] emp_info = new String[_employee_list.Count];
31          for(int i = 0; i<_employee_list.Count; i++){
32           emp_info[i] = _employee_list[i].ToString();
33          }
34          return emp_info;
35        }
36
37        public IEmployee GetEmployeeByEmployeeNumber(String employee_number){
38          IEmployee employee = null;
39          foreach(IEmployee emp in _employee_list){
40            employee = emp;
41            if(employee.EmployeeNumber.Equals(employee_number)) break;
42          }
43           return employee;
44        }
45
46        public IEmployee GetEmployeeByIndex(int index){
47          if((index < 0) || (index >= _employee_list.Count)){ // adjust index
48             index = _employee_list.Count-1;
49          }
50          if(_employee_list.Count > 0) {
51            return _employee_list[index];
52          }
53          return null;
54        }
55
56        public void SortEmployees(){
57         _employee_list.Sort();
58        }
59
60        public void DeleteEmployeeByIndex(int index){
61          if((index < 0) || (index >= _employee_list.Count)){ // adjust index if out of range
```


© 2008 Rick Miller — All Rights Reserved

```
62          index = _employee_list.Count-1;
63        }
64        if(_employee_list.Count > 0) {
65            _employee_list.RemoveAt(index);
66          }
67      }
68
69      public void SaveEmployeesToFile(String filename){
70       if((filename == null) || (filename == String.Empty)){
71          filename = "employees.dat";
72       }
73         FileStream fs = null;
74         try {
75           fs = new FileStream(filename, FileMode.Create);
76           BinaryFormatter bf = new BinaryFormatter();
77           bf.Serialize(fs, _employee_list);
78         }catch(Exception e){
79           Console.WriteLine(e);
80         }finally{
81           if(fs != null){
82             fs.Close();
83           }
84         }
85
86      }
87
88      public void LoadEmployeesFromFile(String filename){
89        if((filename == null) || (filename == String.Empty)){
90          filename = "employees.dat";
91       }
92        FileStream fs = null;
93        try {
94          fs = new FileStream(filename, FileMode.Open);
95          BinaryFormatter bf = new BinaryFormatter();
96          _employee_list  = (List<IEmployee>)bf.Deserialize(fs);
97        }catch(FileNotFoundException fnfe){
98          Console.WriteLine("employees.dat file not found!");
99        }catch(Exception ex){
100         Console.WriteLine(ex);
101       }finally{
102         if(fs != null){
103           fs.Close();
104         }
105       }
106     }
107   } // end Model class definition
108 } // namespace
```

Com.PulpFreePress.View

25.47 View.cs

```
1    using System;
2    using System.Text;
3    using System.Drawing;
4    using System.Windows.Forms;
5    using Com.PulpFreePress.Common;
6
7    namespace Com.PulpFreePress.View {
8      public class View : Form, IView {
9        // constants
10       private const int WINDOW_HEIGHT = 425;
11       private const int WINDOW_WIDTH = 800;
12       private const String WINDOW_TITLE = "Employee Management Application ";
13       private IEmployee _editing_employee = null;
14       private int _editing_employee_index = 0;
15
16       //menu fields
17       private MenuStrip _ms;
18       private ToolStripMenuItem _fileMenu;
19       private ToolStripMenuItem _loadMenuItem;
20       private ToolStripMenuItem _saveMenuItem;
21       private ToolStripMenuItem _exitMenuItem;
22       private ToolStripMenuItem _editMenu;
23       private ToolStripMenuItem _listMenuItem;
24       private ToolStripMenuItem _sortMenuItem;
25       private ToolStripMenuItem _newSalariedEmployeeMenuItem;
26       private ToolStripMenuItem _newHourlyEmployeeMenuItem;
27       private ToolStripMenuItem _editEmployeeMenuItem;
28       private ToolStripMenuItem _deleteEmployeeMenuItem;
```

 © 2008 Rick Miller — All Rights Reserved

```
29
30      // fields
31      private ViewMode _mode = ViewMode.RESTING;
32      private TableLayoutPanel _mainTablePanel;
33      private TableLayoutPanel _infoTablePanel;
34      private FlowLayoutPanel _buttonPanel;
35      private TextBox _mainTextBox;
36      private Label _firstNameLabel;
37      private Label _middleNameLabel;
38      private Label _lastNameLabel;
39      private Label _birthdayLabel;
40      private Label _genderLabel;
41      private Label _hoursWorkedLabel;
42      private Label _hourlyRateLabel;
43      private Label _salaryLabel;
44      private Label _employeeNumberLabel;
45
46      private TextBox _firstNameTextBox;
47      private TextBox _middleNameTextBox;
48      private TextBox _lastNameTextBox;
49      private TextBox _hoursWorkedTextBox;
50      private TextBox _hourlyRateTextBox;
51      private TextBox _salaryTextBox;
52      private TextBox _employeeNumberTextBox;
53
54      private DateTimePicker _birthdayPicker;
55      private GroupBox _genderBox;
56      private RadioButton _maleRadioButton;
57      private RadioButton _femaleRadioButton;
58      private Button _clearButton;
59      private Button _submitButton;
60      private OpenFileDialog _openFileDialog;
61      private SaveFileDialog _saveFileDialog;
62      private bool _createMode;
63
64
65      // public properties -
66      public ViewMode Mode {
67        get { return _mode; }
68        set {
69          _mode = value;
70          this.SetWindowTitleBasedOnMode();
71        }
72      }
73
74      public IEmployee EditingEmployee {
75        get { return _editing_employee; }
76        set {
77          _editing_employee = value;
78          if(_editing_employee.GetType() == typeof(HourlyEmployee)){
79             this.PopulateEditFields((HourlyEmployee)_editing_employee); // ugly baby!
80           }else {
81             this.PopulateEditFields((SalariedEmployee)_editing_employee); // ugly baby!
82           }
83        }
84      }
85
86      public int EditingEmployeeIndex {
87        get { return _editing_employee_index; }
88        set { _editing_employee_index = value; }
89      }
90
91      public String FirstName {
92        get { return _firstNameTextBox.Text; }
93        set { _firstNameTextBox.Text = value; }
94      }
95
96      public String MiddleName {
97        get { return _middleNameTextBox.Text; }
98        set { _middleNameTextBox.Text = value; }
99      }
100
101     public String LastName {
102       get { return _lastNameTextBox.Text; }
103       set { _lastNameTextBox.Text = value; }
104     }
105
106     public DateTime Birthday {
107       get { return _birthdayPicker.Value; }
108       set { _birthdayPicker.Value = value; }
109     }
```

 © 2008 Rick Miller — All Rights Reserved

```
110
111     public String MainTextBoxText {
112       set { _mainTextBox.Text = value; }
113     }
114
115     public Sex Gender {
116       get { return this.RadioButtonToSexEnum(); }
117       set { this.SetRadioButton(value); }
118     }
119
120     public bool CreateMode {
121       get { return _createMode; }
122       set { _createMode = value; }
123     }
124
125     public String Salary {
126       get { return _salaryTextBox.Text; }
127       set { _salaryTextBox.Text = value; }
128     }
129
130     public String HoursWorked {
131       get { return _hoursWorkedTextBox.Text; }
132       set { _hoursWorkedTextBox.Text = value; }
133     }
134
135     public String HourlyRate {
136       get { return _hourlyRateTextBox.Text; }
137       set { _hourlyRateTextBox.Text = value; }
138     }
139
140     public String EmployeeNumber {
141       get { return _employeeNumberTextBox.Text; }
142       set { _employeeNumberTextBox.Text = value; }
143     }
144
145     public bool SubmitOK {
146       set { _submitButton.Enabled = value; }
147     }
148
149     public View(IController externalHandler){
150       this.InitializeComponent(externalHandler);
151     }
152
153     private void InitializeComponent(IController controller){
154       this.InitializeMenus(controller);
155      _mainTablePanel = new TableLayoutPanel();
156      _mainTablePanel.RowCount = 2;
157      _mainTablePanel.ColumnCount = 2;
158      _mainTablePanel.Anchor = AnchorStyles.Top |   AnchorStyles.Bottom | AnchorStyles.Right
159                             | AnchorStyles.Left;
160      _mainTablePanel.Padding = new Padding(10, 50, 10, 10);
161     // _mainTablePanel.Dock = DockStyle.Left;
162      _mainTablePanel.Height = 475;
163      _mainTablePanel.Width = 700;
164      _infoTablePanel = new TableLayoutPanel();
165      _infoTablePanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right
166                             | AnchorStyles.Left;
167      _infoTablePanel.RowCount = 9; // was 2
168      _infoTablePanel.ColumnCount = 2;
169      _infoTablePanel.Height = 300;
170      _infoTablePanel.Width = 425;
171      _buttonPanel = new FlowLayoutPanel();
172      _buttonPanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right | AnchorStyles.Left;
173      _buttonPanel.Width = 500;
174      _buttonPanel.Height = 200;
175
176      _mainTextBox = new TextBox();
177      _mainTextBox.Height = 200;
178      _mainTextBox.Width = 400;
179      _mainTextBox.Multiline = true;
180      _mainTextBox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right | AnchorStyles.Left;
181
182      _firstNameLabel = new Label();
183      _firstNameLabel.Text = "First Name:";
184      _middleNameLabel = new Label();
185      _middleNameLabel.Text = "Middle Name:";
186      _lastNameLabel = new Label();
187      _lastNameLabel.Text = "Last Name:";
188      _hoursWorkedLabel = new Label();
189      _hoursWorkedLabel.Text = "Hours Worked:";
190      _hourlyRateLabel = new Label();
```

 © 2008 Rick Miller — All Rights Reserved

```
191        _hourlyRateLabel.Text = "Hourly Rate:";
192        _salaryLabel = new Label();
193        _salaryLabel.Text = "Salary:";
194        _employeeNumberLabel = new Label();
195        _employeeNumberLabel.Text = "Employee Number:";
196        _employeeNumberLabel.Width = 125;
197
198
199        _birthdayLabel = new Label();
200        _birthdayLabel.Text = "Birthday";
201        _genderLabel = new Label();
202        _genderLabel.Text = "Gender";
203        _firstNameTextBox = new TextBox();
204        _firstNameTextBox.Width = 200;
205        _middleNameTextBox = new TextBox();
206        _middleNameTextBox.Width = 200;
207        _lastNameTextBox = new TextBox();
208        _lastNameTextBox.Width = 200;
209        _hoursWorkedTextBox = new TextBox();
210        _hoursWorkedTextBox.Width = 200;
211        _hourlyRateTextBox = new TextBox();
212        _hourlyRateTextBox.Width = 200;
213        _salaryTextBox = new TextBox();
214        _salaryTextBox.Width = 200;
215        _employeeNumberTextBox = new TextBox();
216        _employeeNumberTextBox.Width = 200;
217
218        _birthdayPicker = new DateTimePicker();
219        _genderBox = new GroupBox();
220        _genderBox.Text = "Gender";
221        _genderBox.Height = 75;
222        _genderBox.Width = 200;
223        _maleRadioButton = new RadioButton();
224        _maleRadioButton.Text = "Male";
225        _maleRadioButton.Checked = true;
226        _maleRadioButton.Location = new Point(10, 20);
227        _femaleRadioButton = new RadioButton();
228        _femaleRadioButton.Text = "Female";
229        _femaleRadioButton.Location = new Point(10, 40);
230        _genderBox.Controls.Add(_maleRadioButton);
231        _genderBox.Controls.Add(_femaleRadioButton);
232        _clearButton = new Button();
233        _clearButton.Text = "Clear";
234        _clearButton.Name = "Clear";
235        _clearButton.Click += controller.UniversalHandler;
236
237        _submitButton = new Button();
238        _submitButton.Text = "Submit";
239        _submitButton.Name = "Submit";
240        _submitButton.Click += controller.UniversalHandler;
241        _submitButton.Enabled = false;
242
243        _infoTablePanel.SuspendLayout();
244        _infoTablePanel.Controls.Add(_firstNameLabel);
245        _infoTablePanel.Controls.Add(_firstNameTextBox);
246        _infoTablePanel.Controls.Add(_middleNameLabel);
247        _infoTablePanel.Controls.Add(_middleNameTextBox);
248        _infoTablePanel.Controls.Add(_lastNameLabel);
249        _infoTablePanel.Controls.Add(_lastNameTextBox);
250        _infoTablePanel.Controls.Add(_birthdayLabel);
251        _infoTablePanel.Controls.Add(_birthdayPicker);
252        _infoTablePanel.Controls.Add(_genderLabel);
253        _infoTablePanel.Controls.Add(_genderBox);
254        _infoTablePanel.Controls.Add(_employeeNumberLabel);
255        _infoTablePanel.Controls.Add(_employeeNumberTextBox);
256        _infoTablePanel.Controls.Add(_hoursWorkedLabel);
257        _infoTablePanel.Controls.Add(_hoursWorkedTextBox);
258        _infoTablePanel.Controls.Add(_hourlyRateLabel);
259        _infoTablePanel.Controls.Add(_hourlyRateTextBox);
260        _infoTablePanel.Controls.Add(_salaryLabel);
261        _infoTablePanel.Controls.Add(_salaryTextBox);
262        _infoTablePanel.Dock = DockStyle.Top;
263
264        _buttonPanel.SuspendLayout();
265        _buttonPanel.Controls.Add(_clearButton);
266        _buttonPanel.Controls.Add(_submitButton);
267
268        _mainTablePanel.SuspendLayout();
269        _mainTablePanel.Controls.Add(_mainTextBox);
270        _mainTablePanel.Controls.Add(_infoTablePanel);
271        _mainTablePanel.Controls.Add(_buttonPanel);
```


© 2008 Rick Miller — All Rights Reserved

```
272        _mainTablePanel.SetColumnSpan(_buttonPanel, 2);
273
274         this.SuspendLayout();
275         this.Controls.Add(_mainTablePanel);
276         this.Width = WINDOW_WIDTH;
277         this.Height = WINDOW_HEIGHT;
278         this.MinimumSize = new Size(WINDOW_WIDTH, WINDOW_HEIGHT);
279         //this.MaximumSize = new Size(WINDOW_WIDTH, WINDOW_HEIGHT);
280         this.Text = WINDOW_TITLE;
281         _infoTablePanel.ResumeLayout();
282         _buttonPanel.ResumeLayout();
283         _mainTablePanel.ResumeLayout();
284         this.ResumeLayout();
285         _openFileDialog = new OpenFileDialog();
286         _saveFileDialog = new SaveFileDialog();
287         this.EnableHourlyFields(false);
288         this.EnableSalaryFields(false);
289         this.SetWindowTitleBasedOnMode();
290
291       }
292
293       private void InitializeMenus(IController controller){
294         // setup the menus
295         _ms = new MenuStrip();
296
297         _fileMenu = new ToolStripMenuItem("File");
298         _loadMenuItem = new ToolStripMenuItem("Load...", null,
299                                                 new EventHandler(controller.UniversalHandler));
300         _loadMenuItem.Name = "Load";
301         _saveMenuItem = new ToolStripMenuItem("Save...", null,
302                                                 new EventHandler(controller.UniversalHandler));
303         _saveMenuItem.Name = "Save";
304         _exitMenuItem = new ToolStripMenuItem("Exit", null, new EventHandler(controller.UniversalHandler));
305         _exitMenuItem.Name = "Exit";
306
307         _editMenu = new ToolStripMenuItem("Edit");
308         _listMenuItem = new ToolStripMenuItem("List", null, new EventHandler(controller.UniversalHandler));
309         _listMenuItem.Name = "List";
310         _sortMenuItem = new ToolStripMenuItem("Sort", null, new EventHandler(controller.UniversalHandler));
311         _sortMenuItem.Name = "Sort";
312         _newSalariedEmployeeMenuItem = new ToolStripMenuItem("New Salaried Employee", null,
313                                                     new EventHandler(controller.UniversalHandler));
314         _newSalariedEmployeeMenuItem.Name = "NewSalariedEmployee";
315         _newHourlyEmployeeMenuItem = new ToolStripMenuItem("New Hourly Employee", null,
316                                                     new EventHandler(controller.UniversalHandler));
317         _newHourlyEmployeeMenuItem.Name = "NewHourlyEmployee";
318         _editEmployeeMenuItem = new ToolStripMenuItem("Edit Employee", null,
319                                                     new EventHandler(controller.UniversalHandler));
320         _editEmployeeMenuItem.Name = "EditEmployee";
321         _deleteEmployeeMenuItem = new ToolStripMenuItem("Delete Employee", null,
322                                                     new EventHandler(controller.UniversalHandler));
323         _deleteEmployeeMenuItem.Name = "DeleteEmployee";
324
325
326         _fileMenu.DropDownItems.Add(_loadMenuItem);
327         _fileMenu.DropDownItems.Add(_saveMenuItem);
328         _fileMenu.DropDownItems.Add(_exitMenuItem);
329         _ms.Items.Add(_fileMenu);
330
331         _editMenu.DropDownItems.Add(_listMenuItem);
332         _editMenu.DropDownItems.Add(_sortMenuItem);
333         _editMenu.DropDownItems.Add("-");
334         _editMenu.DropDownItems.Add(_newSalariedEmployeeMenuItem);
335         _editMenu.DropDownItems.Add(_newHourlyEmployeeMenuItem);
336         _editMenu.DropDownItems.Add(_editEmployeeMenuItem);
337         _editMenu.DropDownItems.Add("-");
338         _editMenu.DropDownItems.Add(_deleteEmployeeMenuItem);
339         _ms.Items.Add(_editMenu);
340
341         _ms.Dock = DockStyle.Top;
342         this.MainMenuStrip = _ms;
343         this.Controls.Add(_ms);
344
345       }
346
347       public void SetWindowTitleBasedOnMode(){
348          switch(Mode){
349            case ViewMode.RESTING: this.Text = WINDOW_TITLE + "- Resting";
350                                   break;
351            case ViewMode.SALARIED: this.Text = WINDOW_TITLE + "- New Salaried Employee";
352                                    break;
```

 © 2008 Rick Miller — All Rights Reserved

```
353           case ViewMode.HOURLY: this.Text = WINDOW_TITLE + " - New Hourly Employee";
354                                  break;
355           case ViewMode.EDIT: this.Text = WINDOW_TITLE + "- Edit Employee";
356                                  break;
357         }
358       }
359
360       public void ClearInputFields(){
361         _firstNameTextBox.Text = String.Empty;
362         _middleNameTextBox.Text = String.Empty;
363         _lastNameTextBox.Text = String.Empty;
364         _maleRadioButton.Checked = true;
365         _birthdayPicker.Value = DateTime.Now;
366         _hoursWorkedTextBox.Text = String.Empty;
367         _hourlyRateTextBox.Text = String.Empty;
368         _salaryTextBox.Text = String.Empty;
369         _employeeNumberTextBox.Text = String.Empty;
370       }
371
372       private Sex RadioButtonToSexEnum(){
373         Sex gender = Sex.MALE;
374         if(_maleRadioButton.Checked){
375           gender = Sex.MALE;
376         }else{
377             gender = Sex.FEMALE;
378           }
379         return gender;
380       }
381
382       private void SetRadioButton(Sex gender){
383         if(gender == Sex.MALE){
384           _maleRadioButton.Checked = true;
385         }else{
386           _femaleRadioButton.Checked = true;
387         }
388       }
389
390       public void EnableSalaryFields(bool state){
391         _salaryLabel.Enabled = state;
392         _salaryTextBox.Enabled = state;
393       }
394
395       public void EnableHourlyFields(bool state){
396         _hoursWorkedLabel.Enabled = state;
397         _hoursWorkedTextBox.Enabled = state;
398         _hourlyRateLabel.Enabled = state;
399         _hourlyRateTextBox.Enabled = state;
400       }
401
402       public void EnableSubmitButton(bool state){
403         _submitButton.Enabled = state;
404       }
405
406       public void DisplayEmployeeInfo(String[] employees_info) {
407         StringBuilder sb = new StringBuilder();
408         foreach(String s in employees_info){
409           sb.Append(s + "\r\n");
410         }
411         _mainTextBox.Text = sb.ToString();
412       }
413
414       public IEmployee GetNewSalariedEmployee(){
415         SalariedEmployee employee = new SalariedEmployee();
416         PayInfo p = new PayInfo();
417         p.Salary = Double.Parse(Salary);
418         employee.PayInfo = p;
419         this.FillInStandardEmployee(employee);
420         return employee;
421       }
422
423       public IEmployee GetNewHourlyEmployee(){
424         HourlyEmployee employee = new HourlyEmployee();
425         PayInfo p = new PayInfo();
426         p.HoursWorked = Double.Parse(HoursWorked);
427         p.HourlyRate = Double.Parse(HourlyRate);
428         employee.PayInfo = p;
429         this.FillInStandardEmployee(employee);
430         return employee;
431       }
432
433       public IEmployee GetEditedEmployee(){
```


© 2008 Rick Miller — All Rights Reserved

```
434        if(EditingEmployee.GetType() == typeof(HourlyEmployee)){
435          return this.FillInEditedEmployee((HourlyEmployee)EditingEmployee);
436        }else{
437          return this.FillInEditedEmployee((SalariedEmployee)EditingEmployee);
438          }
439      }
440
441      public String GetSaveFile(){
442          _saveFileDialog.ShowDialog();
443         return _saveFileDialog.FileName;
444      }
445
446      public String GetLoadFile(){
447        _openFileDialog.ShowDialog();
448        return _openFileDialog.FileName;
449      }
450
451      public int SelectedLineNumber(){
452        int index = _mainTextBox.SelectionStart;
453        int line_number = _mainTextBox.GetLineFromCharIndex(index);
454        return line_number;
455      }
456
457      private IEmployee FillInEditedEmployee(SalariedEmployee employee){
458        employee.PayInfo.Salary = Double.Parse(Salary);
459        return this.FillInStandardEmployee(employee);
460      }
461
462      private IEmployee FillInEditedEmployee(HourlyEmployee employee){
463        employee.PayInfo.HoursWorked = Double.Parse(HoursWorked);
464        employee.PayInfo.HourlyRate = Double.Parse(HourlyRate);
465        return this.FillInStandardEmployee(employee);
466      }
467
468      private IEmployee FillInStandardEmployee(IEmployee employee){
469        employee.FirstName = FirstName;
470        employee.MiddleName = MiddleName;
471        employee.LastName = LastName;
472        employee.Gender = Gender;
473        employee.Birthday = Birthday;
474        employee.EmployeeNumber = EmployeeNumber;
475        return employee;
476      }
477
478      public void PopulateEditFields(SalariedEmployee employee){
479        EnableSalaryFields(true);
480        EnableHourlyFields(false);
481        Salary = employee.PayInfo.Salary.ToString();
482        this.PopulateStandardEditFields(employee);
483
484      }
485
486      public void PopulateEditFields(HourlyEmployee employee){
487        EnableHourlyFields(true);
488        EnableSalaryFields(false);
489        HoursWorked = employee.PayInfo.HoursWorked.ToString();
490        HourlyRate = employee.PayInfo.HourlyRate.ToString();
491        this.PopulateStandardEditFields(employee);
492      }
493
494      private void PopulateStandardEditFields(IEmployee employee){
495        FirstName = employee.FirstName;
496        MiddleName = employee.MiddleName;
497        LastName = employee.LastName;
498        Birthday = employee.Birthday;
499        Gender = employee.Gender;
500        EmployeeNumber = employee.EmployeeNumber;
501      }
502
503    } // end View class definition
504  } // end namespace
```

Com.PulpFreePress.Controller

25.48 Controller.cs

```
1    using System;
2    using System.Windows.Forms;
3    using Com.PulpFreePress.Common;
4    using Com.PulpFreePress.Exceptions;
```

 © 2008 Rick Miller — All Rights Reserved

```
5     using Com.PulpFreePress.Commands;
6     using Com.PulpFreePress.Model;
7     using Com.PulpFreePress.View;
8     using Com.PulpFreePress.Utils;
9
10    public class Controller : IController {
11
12       private CommandFactory command_factory = null;
13       private IModel its_model;
14       private IView its_view;
15
16       public Controller(){
17         command_factory = CommandFactory.GetInstance();
18         its_model = new Model();
19         its_view = new Com.PulpFreePress.View.View(this);
20         Application.Run((Form)its_view);
21       }
22
23
24      public void UniversalHandler(Object sender, EventArgs e){
25        try{
26         BaseCommand command = null;
27         if(sender.GetType() == typeof(Button)){
28           command = command_factory.GetCommand(((Button)sender).Name);
29         }else{
30           command = command_factory.GetCommand(((ToolStripMenuItem)sender).Name);
31          }
32         command.Model = its_model;
33         command.View = its_view;
34         command.Execute();
35         }catch(CommandNotFoundException cnfe){
36            Console.WriteLine("Command not found!");
37         }
38      }
39
40      public static void Main(){
41         new Controller();
42      } // end Main() method
43    } // end Controller class definition
```

Running The Application

You can run the application with the msbuild utility by running the run target of the EmployeeManagement-App.proj file. The run target is the default target and can be run like so:

```
msbuild /t:run
```

...or optionally just by typing...

```
msbuild
```

...at the command line. Figure 25-6 shows the user interface of 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 object's via the .NET runtime'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 consists of one or more classes working together to realize the functionality of a particular application, 2) the *view*,

© 2008 Rick Miller — All Rights Reserved

Figure 25-6: Interacting with the Employee Management Application

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 25.12 through 25.14.

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 C# .NET 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. **UML Diagram:** Draw a UML class diagram for the employee management application presented in the last section of this chapter.

 © 2008 Rick Miller — All Rights Reserved

Suggested Projects

1. **Legacy Datafile Adapter Revisited:** Write an application with the help of the patterns presented in this chapter that accesses data via the legacy datafile adapter class and supporting classes presented in Chapter 17, examples 17.9 through 17.16. 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 20 Projects Revisited:** Revisit any of the suggested projects listed at the end of Chapter 20 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 component 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

© 2008 Rick Miller — All Rights Reserved

Notes

© 2008 Rick Miller — All Rights Reserved

Appendices

© 2006 Rick Miller — All Rights Reserved

© 2006 Rick Miller — All Rights Reserved

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. *This results in a clear problem definition and a list of required project features.*
	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. *This results in a high-level solution statement that can be translated into an application design.*
	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. *This results in a notional understanding of the language features required to effect a good design and solve the problem.*
	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. *This results in a plan of attack!*

Table 26-1: Project Approach Strategy

©2008 Rick Miller — All Rights Reserved

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 26-2: Development Cycle

Final Project Review Checklist

Check-Off	Review	What To Check For
	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.
	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 disk, CD-ROM, or USB memory stick (i.e., removable media)	Ensure all the required source and executable files are present. Try running your project from the removable medium to make sure you have included all the required files.

Table 26-3: Final Project Review Checklist

©2008 Rick Miller — All Rights Reserved

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
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	&	

Table Appendix B-1: ASCII Table

©2008 Rick Miller — All Rights Reserved

Decimal	Octal	Hex	Binary	Value	Comment
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	
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	^	

Table Appendix B-1: ASCII Table

©2008 Rick Miller — All Rights Reserved

Decimal	Octal	Hex	Binary	Value	Comment
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

©2008 Rick Miller — All Rights Reserved

 ©2008 Rick Miller — All Rights Reserved

Appendix C

Identifier Naming: Writing Self-Commenting Code

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 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. C# allows unlimited-length identifier names, so you can afford to be descriptive.

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 *C# 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. This is referred to as *camel case*. Table Appendix C-1 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 26-1: Class Naming Examples

©2008 Rick Miller — All Rights Reserved

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 Appendix C-2 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 26-2: 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 Appendix C-3 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 26-3: Variable Naming Examples

You may want to start field variable names with an underscore character to make them easy to spot in your code.

Method Names

A method represents a named series of 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 an upper-case character, and capitalize each subsequent word of a multiple-word method. The only exception to this naming rule is for constructor methods, which must be exactly the same name as the class in which they appear. Table Appendix C-4 gives some examples.

Method Name	Comment
`PrintFloor`	Multiple-word method name. The first word is an action word.
`SetMaxValue`	Multiple-word method name. This is an example of a mutator method.
`GetMaxValue`	Another multiple-word method name. This is an example of an accessor method.
`Start`	A single-word method name.

Table 26-4: Method Naming Examples

©2008 Rick Miller — All Rights Reserved

Property Names

The property name should accurately reflect the nature of the property. Table Appendix C-5 offers several examples.

Property Name	Comment
`StudentCount`	Multiple-word property name.
`MaxValue`	Multiple-word property name.
`FirstName`	Another multiple-word property name.
`Length`	A single-word Property name.

Table 26-5: Property Naming Examples

©2008 Rick Miller — All Rights Reserved

©2008 Rick Miller — All Rights Reserved

Index

Symbols

- ! 122
- - 122
- -- 122
- != 125
- #define DEBUG directive 645
- #endregion directive 541
- #region directive 541
- % 123
- * 123
- + 122
- ++ 122
- += operator 304
- .NET Framework
  - downloading 21
  - installing 20–21
- .NET Remoting
  - Singleton mode 468
- .NET remoting 466–477
  - configuration files 472
  - network communication handled by 469
  - passing collection of Person objects between remote object and client 474
  - persisting remote object state 469
  - purpose of 466
  - registering channels 468
  - registering service name 468
  - registering well known service types 468
  - remote object access via interface 470
  - serializing complex objects 474
  - simple example 467
  - SingleCall mode 468
  - SingleCall vs. Singleton remote object modes 469
  - swapping remote objects
    - enabling with interfaces 470
  - three primary channels 467
  - three required components 466
- .NET Remoting Architecture 466
- .NET remoting infrastructure 495
- / 123
- /d
  - DEBUG compiler switch 646
- ; 119
- < 125
- << 124
- <= 125
- = 119
- == 125
- > 125
- >= 125
- ~ 122

A

- abstract
  - classes 267
  - methods 267
- abstract class 257
  - expressing in UML 268
  - purpose of 268
  - term defined 257
- abstract class vs. interface 270, 271
- abstract data types 191
- abstract keyword
  - using to declare classes and methods 269
- abstract methods
  - implementing in derived classes 269
- abstract thinking 9
- abstraction
  - problem 9
  - the art of programming 190
- abstractions
  - selecting the right kinds of 662
- access
  - horizontal 274
  - vertical 274
- Access Control Graph (ACG) 675
- access modifiers 201
  - default/package 201
  - most often used 274
  - private 201
  - protected 201
  - public 201
- address bus 81
- addressing local machine 455
- ADO.NET 494
- aggregation 234, 235, 237, 250
  - aggregate constructors 236
  - composite 236, 250
  - composite example code 239
  - definition 235
  - determining type by who controls object lifetime 236
  - effects of garbage collector 236
  - example
    - engine simulation 242
    - engine simulation class diagram 244
  - simple 236, 237, 250
  - simple example code 238
  - two types of 676
- algorithm
  - running time 85
  - understanding the concept of 76
  - working definition of 83
- algorithm growth rate 85
- algorithms 76, 83
  - good vs. bad 83
- alter statement
  - used to create foreign key constraint 512
- analysis 48, 668
- Ansel Adams 668
- API Framework
  - blessing and curse 94
- API reference documentation
  - class general overview page 96
  - class member page 97
  - obsolete APIs 102
  - Syntax section 101
- API reference information
  - definitive source 94
- application
  - definition 111
  - graceful recovery 46
  - layers 454
  - physical deployment 454
  - physical tier distribution 456
  - simple
    - structure 111
  - tier responsibilities 456
  - tiers 454
- Application class
  - Run() method 293
  - use of to run GUI programs 293
- application distribution 454
  - across multiple computers 455
- application domain 384
- application layer 458

© 2008 Rick Miller — All Rights Reserved

application layers 495
application message loop 293
application tiers
- logical 456
- separation of concerns 456

ApplicationException 367
applications
- multitiered 456

architectural diagram
- multitiered database application 494

architecture
- flexibility 669
- modularity 669
- reliability 669
- stability 669

array 339, 342
- creating with literal values 168
- declaration syntax 163
- definition of 162
- difference between value type and reference type arrays 169
- dynamic resizing
  - example code 339
- elements 162
- functionality provided by array types 164
- homogeneous elements 162
- Main() method String parameter 181
- multidimensional 176, 179
- of value types 166
- properties of 165
- references
  - calling Array class methods on 167
- single dimensional 166
- single dimensional in action 171
- specifying length 163
- specifying types 163
- two dimensional
  - example program 179
- type inheritance hierarchy 164
- value type
  - memory arrangement 166

Array class 182
array initializer expression 178
array literal 168, 169
array of arrays 178
array processing 46
array-based collection
- growing on insertion 339

arrays 162
- rectangular 176
- sorting with Array class 182
- two-dimensional
  - processing 62
- using to solve problems 162

Ashmore's hash code algorithm 631
assembly
- definition 111

Assertion Failed dialog 646
association 235, 250
- definition 235

associativity
- operator 121
  - forcing 121

asynchronous method calls 400
asynchronous methods
- EndInvoke() method 402
- IAsyncResult interface 403
- obtaining results from 402
- providing callback method to BeginInvoke() method 402

attribute candidates 47
attributes 45
automated water tank custom event example 326–331
auxiliary storage device 410

B

BackgroundWorker 382
BackgroundWorker class 396
BackgroundWorker events 396
bad software architecture
- characteristics of 661

base class
- methods
  - overriding 266
- source code example 259

Base Class Libraries (BCL) 100
BaseCommand class 698
BaseDAO
- class definition
  - using DatabaseFactory class 522

behavior
- generalized 256

behavior contract 615
Bertrand Meyer 656, 671
Bertrand Meyer's Design by Contract (DbC) 643
binary data 420–422
BinaryFormatter class 414, 486
BinaryReader class 420, 422, 423
BinaryWriter class 420, 422
bit 80, 81
BitMap class 299
- using to create Image object 300

Bloch's hash code algorithm 631
block 479
blocking I/O operation 479
Bounds
- data that comprises 298
- property
  - printing to screen 298

Bounds property 298
- setting example 302

boxing 225
break 142
bridge 451
Business Layer 495
business object
- definition 495

business objects 494, 495
business rules 495
- creep 495

Button 291
byte 80, 81

C

C# compile and execute process 86
cache memory 80
calling base class constructor with base() 260
camel case 199, 733
cascade delete 502
- SQL
  - cascade delete
    - testing 515

casting 264, 351
- advice on use of 265

chained hash table 345
character constants
- declaring
  - example 55
- using in switch statement 56

Christopher Alexander 688
class 111, 257
- abstract 267
  - expressing in UML 268
  - purpose of 268
- abstract class 257
- four categories of members 194
- non-static fields 195
- sealed 274
- static fields 195
- term definition 257

 © 2008 Rick Miller — All Rights Reserved

class declarations
viewed as behavior specifications 644
class definition
adding fields 207
adding instance methods 208
constructor method 208
starting 207
class invariant 644, 646
defined 644
class invariants 644
class member access
default when omitting access modifier 274
classes
classes vs. structs 225
number in an application 234
Class-Wide Fields 195
Click event 303
client 450, 453
application 450, 453
hardware 450, 453
client application 466
client coordinates 299
client-server applications
See also TCP/IP client-server
TCP/IP 478
with .NET remoting 466
cloning objects 627
CloseReader() method 523
Coad's Inheritance Criteria 672
code blocks
executing in if statements 139
code library
creating 86
code module
creating 86
code reuse 668
coding convention
adopting 733
cohesion 15, 203
collateral roles
modeling 674
collections
ArrayList
usage example 340
casting 351
extending ArrayList 352
extracting elements into arrays 361
general characteristics 338
generic
example code 354–356
KeyedCollection<TKey, TItem> example 355
List<T> 354
IComparer<T, T> 359, 636
implementing IComparable<T> 357, 634
interfaces 338
linked list node elements 343
making an object sortable 357, 634
non-generic to generic mapping table 349
old-school style 350–353
old-school style programming 348
performance characteristics
arrays 342
hashtable 345
linked list 343
Person list example 351
red-black tree node elements 346
sorting 357
rules for implementing IComparable<T>.CompareTo() method 358, 635
specialized 349
underlying data structures 349
using foreach to iterate over
example 351
Color structure 299
columns 501
command console layout properties
modifying 28
command pattern 688, 697
CommandFactory class 699
command-line arguments
processing 181
command-line compiler 20
command-line tools 20
why you should learn 20
Common Language Infrastructure
four parts 87
Common Language Infrastructure (CLI) 86, 87
Common Language Runtime (CLR) 90
Common Language Specification (CLS) 88
Common Log File System 438
Common Type System (CTS) 88
compiler errors
dealing with 30
finding their meaning on MSDN 30
fixing 14
compiling
simple application 111
compiling multiple source files 234
compiling source file
how to 29
compiling with csc
using target switch example 215
complex application behavior 234
complex project folder organization 516
complexity
conceptual 14, 234, 235, 250
managing physical 15
physical 15, 234, 235, 250
relationship between physical and conceptual 15
Component 293
components
adding to Controls collection 302
adding to windows 301
initializing in separate method 302
composite aggregation
defined 236
composition 668, 676
as force multiplier 676
compositional design 234, 676
compositionists 668
computer
architecture
feature set 79
feature set accessibility 79
feature set implementation 79
three aspects of 79
definition of 76
memory
organization 79
processing cycle 82
system 76
components of 77
hard drive 77
keyboard 77
main logic board 77
memory 77, 80
monitor 77
mouse 77
processor 77
speakers 77
system unit 77
vs. computer system 76
computer network
definition 450
purpose 450
computer program
modeling real world problem 190

© 2008 Rick Miller — All Rights Reserved

computers 76
conceptual complexity 14, 234
managing 14
taming 14
concrete class 260
concurrently executing applications 384
condition
exception 366
configuration file
example 528
configuration files
.NET remoting 472
configuration-management tool 15
connection pooling 495
connection string
database
configuration file setting 499
console applications 110–131
console text color
changing
example code 483
console text menu
processing user commands
example 56
console text menus
example 53
const 197
constant 47, 195
constants 197
constraint
database 504
constructor methods 206
constructors 616
ContainerControl 293
containing aggregate 237
containment
by reference 236
by value 236
polymorphic 676
contains 237
continue 152
Control 293
control bus 81
controller 695
controls
dynamic layout of 308
registering event handler methods 303
Controls collection
use of 302
coordinates
client 297
origin 298
screen 297
(x,y) pairs 297
origin 298
pixel as basic unit of measure 297
window 297
window placement upon screen 297
copy constructor 625
coupling 15
create tables SQL script 504
creativity
and problem abstraction 190
cross platform
promise of 89
CRUD operations
database
CRUD operations 523
csc
compiling entire source directory 235
compiling multiple source files 234
csc compiler
locating 21
current position 48
custom event
recursive example 327–329
custom events 322
suggested naming convention 331
custom exceptions 374
custom serialization 618, 620

D

DAO layer
building 519
Data Access Layer 495
data access object
definition 495
data access objects 494, 495
data base
key factor in business rules 495
data bus 81
Data Control Language 502
Data Definition Language 502
data link layer 458
Data Manipulation Language 502, 506
data type 47
reference 164
value 164
data types
array 164
SQL Server 505
database
automatically inserting primary key 512
cascade delete 502
columns 501
constraint
definition of 504
converting binary data into bitmap image 526
creating related table with script 511
DataBase.AddInParameter() method 526
foreign key 501, 511
foreign key constraint
naming 512
inserting image data
example code 525
inserting test data into related table 512
inserting value objects into 526
join operation 511
primary key 501
record 514
referential integrity 501
rows 501
table 501
database application
compiling 500
database connection
established via DatabaseFactory 495
database connection string 499
database connection test application 499–500
database management system 501
Database object 499, 523
database script
running
example 504
DatabaseFactory 495, 523
configuration file 499
example code 499
DataBindingComplete event 580
dataConfiguration
configuration file section 499
datagrams 459
DataGridView 562, 564
clicking on row to yield row index 562
data binding 580
DataSource property 562
row index value 562

© 2008 Rick Miller — All Rights Reserved

DateTime structure
- example of use 310

DateTime.Now 310
DbC 643
DbCommand 526
DBMS 501
DbType enumeration
- .NET type mapping table 527

Debug.Assert() method 645
deep copy 614
- defined 624

default class member access 274
default constructor 200
delay
- example code 330

delegate 291, 322
- event subscriber list 322
- EventHandler 323
- method signature specification 323

delegate object
- purpose of 322

delegate type
- purpose of 303
- specification of method signature 304

delegates
- EventHandler 303
- MouseEventHandler 303
- PaintEventHandler 303
- running asynchronous methods with 400

delete command
- SQL
  - commands
    - delete 510

delimiter
- text file 418

Department of Defense 452
dependencies
- managed 669

dependency 194, 235
- definition 235
- effects of dependency relationships between classes 235

dependency inversion principle 661
dependency relationship 250
dependency vs. association 235
deprecated members 201
derived class
- source code example 260

deserialization
- object 414

deserialize
- object
  - from XML file 416

design 668
design by composition 234
Design by Contract 643
design pragmatists 668
development cycle 43
- application 51
- applying 43
- code 43, 728
- creating feature implementation lists 51
- deploying 43
- integrate 43
- iterative application 51
- plan 43, 728
- refactor 43
- test 43, 728
- using 43

development environment
- configuring 22

device driver 410
difference between abstract class and interface 270
difference between readonly and const fields 197
direct base class 194
direction 47
directory
- definition 411

Directory class 411
- example code 412

DirectoryInfo class 411
disk
- driver software 410

distributed applications 450
DockStyle enumeration 309
- values 309

documentation generation 72
dominant roles
- modeling 674

Doxygen 72
Dr. Barbara Liskov 643
Dr. Bertrand Meyer 643
drive letters 411
driver
- creating test code 209

dynamic class loading
- example code 699

dynamic factory pattern
- advantages of 695

dynamic link library 86
dynamic polymorphic behavior 656
DynamicArray
- case study 338

E

ECMA - 335 87
effects of change
- predicting 669

Eiffel 643
EmployeeDAO 495
empty statement 119
Encapsulation 9
encapsulation 201
EndInvoke() method 402
engineering trade-off 668
Enterprise Library Configuration tool 499
Enterprise Library Data Access Application Block 495
entry point 111
enumerated type 59
environment variable 20
environment variables 22–24
Erich Gamma 689
error checking 46
error conditions
- program
  - handling 137
- that cause exceptions
  - examples of 366

errors
- compiler 14

Ethernet 459
event 322
event arguments
- example code 324

event consumer 322
event driven programs 293
event handler
- explicit call to
  - example 580

event handler methods
- registering 303

event handlers
- located in different objects 305

event producer 322
event subscriber list 322
events 200, 303
- and their delegate types
  - table of 303
- BackColorChanged 303
- BackgroundImageChanged 303
- Click 303
- DoubleClick 303

© 2008 Rick Miller — All Rights Reserved

GotFocus 303
GUI
handling in separate object example 307
handled in separate objects 305
MouseClick 303
MouseDoubleClick 303
MouseDown 303
MouseEnter 303
MouseLeave 303
MouseMove 303
MouseUp 303
Paint 303
registering event handler method
example of 304
Exception
class hierarchy 367
public properties 370
exception
definition 366
exception information table 367
exceptions 366–376
catch block 366
catching multiple exceptions
rule of thumb 372
catching with try/catch block 138
CLR handling mechanism 366
custom 374
extending Exception class 374
using throw keyword 375
determining what a method may throw 369
documenting 376
fault handler code 366
low-level to high-level translation 425
purpose of 366
runtime vs. application 368
translating low-level to high-level 375, 425
try block 366
try/catch/finally blocks 371–374
using multiple catch blocks 372
executing application
how to 30
executing SQL command
example code 499
extension inheritance
complications from using 675
vs. functional variation 675

F

façade 688
factory 688
factory class
interfaces involved to employ 674
fault handler code 366
Fields 195
fields
readonly
initializing static readonly fields in static constructor 195
readonly vs. const 197
file
definition 410
File class 411
file I/O 410–443
file position pointer 421, 422
File Transfer Protocol 458
FileDialogs
using 440–442
FileInfo class 411
example code 412
files
manipulating 411–413
FileStream class 414, 422
final project considerations
checklist 66
finalizers 200
First-In-First-Out (FIFO) 347
fixed-length records 422
reading
example code 429
floor 48
flow 11
achieving 12
concept of 11
stages 12
flow charts 59
FlowDirection enumeration
values 309
FlowLayoutPanel 291, 308
properties
AutoSize 309
AutoSizeMode 309
Dock 309
FlowDirection 309
WrapContents 309
purpose of 308
folder 411
folder options
setting 25
foreign key 501, 511
foreign key constraint 512
Form 291, 292, 294
class inheritance hierarchy 292
properties
Backcolor 299
BackgroundImage 299
manipulating 299
simple form program 293
Text property 293
window types created with 292
formatting
numeric strings
table 183
source code 66, 728
from clause
use to join tables
SQL
from clause
use to join tables 514
functional decomposition 8
fundamental language features 46

G

gate 688
gateway 451
generalization
expressing in UML 258
generalized behavior
specifying 256
GetRegisteredWellKnowClientTypes() method 473
good design
goals of 669
good software architecture
characteristics of 662
goto 153
graphical user interface programming 292–318
guarded region
of try block 138
GUI
coding rhythm 317
data input dialog design 570
loading image in PictureBox
example code 528
opening image file with OpenFileDialog
example code 528
separating code from event handlers 305–307
using dialogs to enter data 570
GUI layout
using mock-up sketch to design 559

© 2008 Rick Miller — All Rights Reserved

guillemet characters 194

H

hard disk 410
hardest thing about learning to program 4
has a 237
hash code
  algorithm 631
hash function 345
hash table 342
  chained 345
  open address 345
  slot probe function 345
Height property 302
homogeneous data types 162
horizontal access 201, 274, 616
host 453
HttpChannel 467
Hypertext Transfer Protocol 458

I

ICloneable 627
IDataReader 527
IDE 20
identifier 114
  class name examples 733
  constant name examples 734
  method name examples 734
  naming 114, 733
  variable name examples 734
identifiers 115
  forming 114
if/else statement 140
Image
  converting to byte array 526
image
  using to set Form BackgroundImage property 300
Image class 299
Image data
  storing and transferring as byte array 565
IMessageFilter
  implementation example 296
implementation approach 51
implicit cast 352
indexer
  example code 339
indexers 200
IndexOutOfRangeException
  handling 57
infinite loop 146
inheritance 668, 670–673
  first purpose of 256
  good reasons for using 670
  Meyer's Taxonomy 671
  object-oriented programming with 256
  second purpose of 257
  simple example 259
  third purpose of 257
  three purposes of 256
  valid usage checkpoints 672
inheritance form
  constant 672
  extension 671
  facility 672
  functional variation 672
  implementation 672
  machine 672
  model 671
  reification 672
  restriction 671
  software 672
  structure 672
  subtype 671
  type variation 672
  uneffecting inheritance 672
  variation 672
  view 672
inheritance hierarchy
  assessing with Coad's criteria 673
  navigating 101
inheritists 668
inner join 514
instance constructors 199
integral type size
  be aware of 123
integrated development environment 20
interface 257
  authorized members 257, 270
  purpose of 270
  reducing dependencies with 674
  role of 674
  term definition 257
interface members
  mapping to abstract members 275
interfaces 668
  expressing in UML 271
Intermediate Language (IL) 86
internal 201, 258, 261, 274
Internet Protocol (IP) 459
Internet protocol layers 457
Internet Protocols 452
inter-process communication 467
IP 459
IP address
  parsing with IPAddress.Parse() method 484
IP addresses 459
IPAddress.Parse() method 484
IpcChannel 467
  purpose of 467
is a relationship
  implementing 257
iteration
  development 43
iterative development 43

J

John Vlissides 689
join operation 511
Just-In-Time (JIT) compiler 86

K

keyword
  using as identifier
    example 114
keywords
  reserved
    listing 113

L

Label 291
language features 42, 51, 727
language-features strategy area 48
Last-In-First-Out (LIFO) 347
layout managers 307–312
legacy datafile adapter 422
library
  creating with compiler
    example 467
  referencing with compiler switch
    example 468
linked list 342
Liskov Substitution Principle
  relationship to Meyer Design by Contract Programming 643
  three rules of 654
Liskov Substitution Principle (LSP) 643
List<T>

© 2008 Rick Miller — All Rights Reserved

example code 341
Local Area Network 450
localhost 455
Location property 302
lock keyword 425
compared to Monitor.Wait()/ Monitor.Exit() 425
log files 438–440
loops 145
LSP 643
LSP & DbC
C# support for 643
common goals 643
designing with 644

M

machine code 79, 86
Magic Draw UML Design Tool 241
Main method 110
main method
purpose 112
signatures 112
managed code 89
managed threads 385
MarshalByRefObject 293
use to create remotable object 466
marshaling
remoting method calls 467
MemberwiseClone() 627
memory
address bus 81
alignment 81
bit 80, 81
byte 80, 81
cache 80
control bus 81
data bus 81
hierarchy 80
non-volatile 80
organization 79
RAM 80
ROM 80
volatile 80
word 80, 81
menu 47, 559
menus 312–315
adding submenu items to menus 313
item naming conventions 313
menu item separator
adding 313
menuitems
registering event handlers with 313
MenuStrip
docking to window 313
importance of adding last 313
MenuStrip class 312
ToolStripMenuItem 312
MenuStrip 312
declaring and creating 313
message categories 295
message filters
adding 296
message loop
window 294
message pump 294
message queue 294
message routing
windows 294
messages
system
how they are generated 294
Metadata 88
method
cohesion 203
definition structure 203
parameter list 111
sealed 274
signature
definition 112
method stubbing 13
methods 46, 199, 202
abstract 267
body 205
constructors 206
example definitions 205
local variable scoping 224
modifiers 203
name 205
naming 203
overloading 206
parameter behavior 219
parameter list 205
passing arguments to 219
return types 204
signatures 206
using return values as arguments 224
methods rule 655
Microsoft Build 235
Microsoft Developer Network (MSDN) 20, 94
Microsoft Enterprise Library
installation 498
support for application layers 495
Microsoft Enterprise Library Application Blocks 494
Microsoft Intermediate Language (MSIL) 87
Microsoft SQLServer Express Edition 494
Microsoft Visual C# Express 20
MinuteTick custom event example 323–325
model 45, 695
modeling 45
collateral roles 675
dominant roles 674
dynamic roles 675
model-view-controller 688
model-view-controller (MVC) 695
module
creating with compiler 111
definition 111
monalphabetic substitution 173
Monitor class
synchronizing thread access with 424
usage 424
MSBuild 235, 516
<Csc> task 519
<ItemGroup> tag 518
<project> tag 518
<PropertyGroup> tag 518
<Target> tag 518
compiling value object target 522
default target 519
items
referencing 518
project file
example 517
properties
referencing 518
targets
defining 518
using to manage and build project 517
MSDN 20, 94
MSIL Disassembler 87
multithreaded programming 382
multithreaded server 480
multithreaded server application 454
multithreaded TCP/IP server 480–482
multithreaded vacation 382
multi-tier projects
recommended approach 519
multitiered applications 450, 456
multitiered database application
design 494

© 2008 Rick Miller — All Rights Reserved

multitiered database applications 494–583
MVC 695, 697
Controller
using factory pattern 698
simple example of 696

N

namespaces 7
naming conventions
for custom events 331
nested type 200
nested type declarations 200
network
definition 450
homogeneous vs. heterogeneous 451
purpose 450
network application
layers 454
physical deployment 454
tiers 454
network applications 450
network clients
running multiple on same machine 454
network layer 458
network stream
flushing after writing serialized object 486
network streams
StreamWriter.Flush() method 480
StreamWriter.WriteLine() method 480
networking 450
networking protocols
role of 451
NetworkStream 486
NonSerialized 618
NotePad++ 22
noun 47
noun lists
suggesting possible application objects 46
nouns 46, 47
mapping to data structures 47
numeric formatting 183

O

Object 293
object
cloning 627
their associated type 257
object attributes 46
object behavior
comparison/ordering 615, 634
copy/assignment 614, 623
defined 614
equality 615, 629
fundamental 614, 616
object creation
with System.Activator.GetObject() method 469
object equality 614
object usage scenario evaluation checklist 615
Object.Equals() method
rules for overriding 630
Object.GetHashCode() method
general contract 630
object-oriented analysis 668
object-oriented architecture
extending 642
preferred characteristics 642
reasoning about 642
understanding 642
object-oriented design approach 9
object-oriented programming 190
object-oriented programming enablers 668
object-oriented programming patterns 307
objects
operations upon 257
value vs. reference assignment 624
well-behaved 614
obsolete Thread methods 389
OCP 656
defined 656
example 656
octets 458
OnDeserialized 618
OnDeserializing 618
OnSerialized 618
OnSerializing 618
open address hash table 345
open-closed principle 656
achieving 656
operands 121
operating system
file management services 410
operator associativity 121
operator overloading 200, 590–611
assignment operators 610
binary * / operators 599
binary + - operators 597
binary operators 597
bitwise & | operators 601
comparison operators 603
implicit and explicit cast 607
in the context of your design 590
purpose for 590
table of overloadable operators 590
true false operators 593
unary - operator 591
unary ! operator 592
unary + operator 591
unary ++ -- operators 595
unary operators 591
operator precedence 121
operator semantics 590
operators 120–131, 200
additive 124
assignment 130
conditional AND 129
conditional OR 129
equality 125
logical AND 126
logical OR 126
logical XOR 126
modulus 123
multiplicative 123
overloading 590
primary 121, 122
relational 125
shift 124
ternary 129
type testing 125
unary 122
OptionalField attribute 618
origin 298
overloaded operators
leading to cleaner code 590
overloading 199
override
keyword used to override base class methods 267
overriding
base class methods
enabling with virtual keyword 266
overriding Object.GetHashCode()
checklist 630

P

packet 452

© 2008 Rick Miller — All Rights Reserved

packet-switched network 457
parameter 111
parameter arrays
  example 223
ParameterizedThreadStart delegate 390
  used in multithreaded server 482
parameters
  behavior of reference types 220
  behavior of value-types 220
  how arguments are passed to methods 220
  out parameter modifier 223
  parameter arrays 223
  passing ref arguments 219
  ref keyword 219
params keyword 223
part objects 236
pass by reference 219
pass by value 219
PATH 20
path
  absolute 411
  definition 411
  relative 411
Path class 411
patterns
  command 688
  façade 688
  factory 674, 688
  MVC 688
  singleton 674, 688
pen 47
Peter Coad 672
physical complexity 15, 234
physical layer 458
Point structure 301
  using to place components 301
polymorphic behavior
  example of 267
polymorphic containment 676
polymorphic substitution 674
polymorphism 668
  applied 675
  defined 275, 675
  goal of programming with 675
  planning for proper use of 675
port 468
postcondition 646
  defined 645
postconditions 644
  changing in derived class methods 652
precondition 218, 646
  defined 644
preconditions 644
  changing preconditions of derived class methods 648
  weakening 648
predefined types 115
preempted 384
PreFilterMessage() method 296
prepared statements 526
primary key 501
  automatically incrementing integer 511
private 258, 274
problem abstraction 9, 190
  and the development cycle 191
  end result of 191
  mantra 190
  performing problem analysis 191
  process of 190
problem domain 8, 42, 46, 51, 727
procedural-based design approach 8
process 382
  definition 383, 384
  multithreaded
    definition 384
  single-threaded
    definition 384
processing cycle 82
  decode 82, 83
  execute 82, 83
  fetch 82, 83
  store 82, 83
processor
  block diagram 78
  CISC 78
  machine code 79
  RISC 78
production coders vs. design theorists 669
program
  computer perspective 82
  definition of 82
  human perspective 82
  two views of 82
  what is a C# 110
program control flow statements 136
programming 4
  challenges & frustrations 4
  skills required 4
programming as art 4
programming cycle 12
  code 12
  integrate 12
  plan 12
  refactor 13
  repeating 13
  summarized 13
  test 12
programs 76
  why they crash 83
project approach strategy 7
  application requirements 8
  design 8
  in a nutshell 10
  language features 8
  problem domain 8
  strategy areas 8
project complexity
  managing 14
project folder
  creating 25
project objectives 45
project requirements 8, 51
project specification 47
properties 198
  creating a calculated property 210
  example 208
  get accessors 198
  instance 198
  read-only 198
  read-write 198
  set accessors 198
  static 198
properties rule 655
protected 258, 261, 274
protected block 366
protected code 371
protected internal 201, 258, 261, 274
protocol stack 457
proxy
  used by remoting client 467
pseudocode 59, 60
public 111
public interface 201
publisher 322
  responsibilities 322

Q

quality without a name 688
queue 347
  FIFO characteristic 347
QWAN 688

R

ragged array 178

© 2008 Rick Miller — All Rights Reserved

Ralph Johnson 689
random access file I/O 422–437
  calculating fixed-length record count 422
RDBMS 501
Readonly Fields 195
readonly instance fields 195
readonly static fields 195
readonly vs. const fields 197
realization 271
  expanded form 271
  expressing in UML 271
  lollipop diagram 271
  simple form 271
record 514
record locking 424
Rectangle structure 301
rectangular arrays 176
recursion
  example 329
red-black binary tree 342
refactor 257
refactoring a design 257
reference equality vs. value equality 629
reference parameters 219
reference semantics 225
reference to object combinations 261
reference types 115
referential integrity 501
regression testing
  example 58
relational database 494, 501
relational database management system 501
relationships
  between database tables 501
reliable object-oriented software
  creating 643
remotable object 466
  how to create 466
remote object
  creating for multitiered application 551
Remoting exception
  problem sending bitmap across application domains 563
remoting infrastructure 466, 469
requirements 8, 42, 727
  gaining insight through pictures 47
requirements gathering 8
resource sharing 450
ResumeLayout() method
  purpose of 309
Richard Helm 689
Robert's Rules of Order 451
robot rat project specification 44
  analyzing 45
root directory
  definition 411
routing tables 459
rows 501

S

screen coordinates 297, 299
ScrollableControl 293
sealed class 274
sealed method 274
segments 458
select command 507
selection statements 136
self-commenting code
  writing 733
semantics
  pre and postfix increment and decrement operators 596
  value vs. reference 225
sensor
  multimode
    example 327
serializable attribute 413
serialization
  custom 618, 620
  object 413
serializing
  List<People> to NetworkStream 486
  objects
    as XML 416
serializing objects 413–418
  steps to 414
server 450, 453
  application 450, 453
  hardware 450, 453
  multithreaded 454
  treated as capital equipment 453
server application 466
service 383
shallow copy 614
  defined 624
shallow vs. deep copy 624
shortcut
  creating 26
  modifying properties 27
  modifying start-up folder 27
signature
  method 199
signature rule 655
simple aggregation
  defined 236
simple vs. composite aggregation 236
simplification
  of real-world problems 190
SingleCall 468, 469
single-threaded vacation 382
Singleton 468, 469
singleton 688
socket 478
software design 192
software design patterns 688
  abstract factory 693
  background 688
  command 697
  definition 688
  dynamic factory 693
  factory 693
  factory method 693
  Singleton 690
  specification template 689
Software Development Kit (SDK) 87
software development roles 6
  analyst 6
  architect 7
  programmer 7
sorting
  arrays with Array class 182
  collections 357, 634
source code
  file header 66, 728
  formatting 66, 728
specialization
  expressing in UML 258
SplitContainer
  example code 441
SQL 500–516
  AND operator 515
  commands
    alter 502
    create 502
    delete 506
    drop 502
    insert 506
    select 506, 507
    update 506
    use 502
  constraint
    definition of 504
  creating tables 504

 © 2008 Rick Miller — All Rights Reserved

Data Control Language 502
Data Definition Language 502
Data Manipulation Language 502, 506
database script
dropping and creating tables with 503
database scripts
using 503
executing commands with go 503
from clause 507
inner join 514
join operation 514
order by clause
example 515
prepared statements 526
three sub languages 502
where clause 507
SQL command parameters 526
SQL command parameters and prepared statements
generalized steps 526
SQL command utility
use of 502
-W switch 514
SQL query string constants 526
SQL Server
changing to master database 503
data types 505
four default databases 502
identity operator 512
newid() function 513
use of 508
SQL Server Management Studio 512
installation 496–497
SQLServer Express
installation 495–496
stack 347
LIFO characteristic 347
state transition diagrams 60
statement
for
personality of 148
nineteen kinds of 119
statements 119–131
break 151
chained if/else 141
continue 151, 152
control flow 136
do/while 146
personality of 147
empty 119
executing consecutive if 139
for 148
relationship to while 148
goto 153
if 136
if/else 136, 140
iteration 145
nesting 149
mixing selection and iteration 150
nineteen kinds of 120
selection statements 136
switch 136, 142
condition expression types 142
nested 144
using break in 142
table of 154
try/catch 138
while 145
personality of 145
state-transition diagrams 59
static 114
static constructor 195
static constructors 200
strategy
project approach 7
StreamReader class 418
StreamReaders
use in network programming 450
StreamWriter class 416, 418
strengthening preconditions 650
String
array of 172
string 114
string characters
accessed using array notation 175
string formatting 183
structs
advice on when to use 227
authorized members 226
behavior during assignment 226
behavior of this 226
boxing and unboxing 226
default field values 226
Structured Query Language 500–516
structures
structures vs. classes 225
stubbing 13
subfolder 411
subject matter experts 8
subscriber 322
responsibilities 322
subscriber notification process 323
supertypes & subtypes
reasoning about 643
SuspendLayout() method
purpose of 309
switch
implicit case fall-through 143
switch statement 142
system message queue 294
System namespace
exploring 96
System.Activator.GetObject()
example code 469
System.Collections 348
System.Collections.Generic 348
System.Collections.ObjectModel 349
System.Collections.Specialized 349
System.Diagnostics namespace 645
System.Guid
use of as primary key 522
System.ValueType class
direct base class for all value types 118
SystemException 367

T

table 501
TableLayoutPanel 291, 310
adding multidimensional array of controls to 311
properties
ColumnCount 311
RowCount 311
TCP 458
octet sequencing 458
TCP/IP 450, 451, 452, 457–460
application layer 458
data link layer 459
network layer 459
physical layer 459
transport layer 458
TCP/IP client server programming 478–489
TCP/IP client-server
binding TcpListener object to machine IP address and port 479
calling TcpListener.AcceptTcpClient() method 479
calling TcpListener.Start() method 479
connection process illustrated 478
listening on multiple IP addresses 482
multithreaded server
building 480

© 2008 Rick Miller — All Rights Reserved

serializing complex objects between 484
simple example code 479
TcpChannel 467, 468, 469
TcpClient 478
TcpListener 478
TELNET 458
test data
inserting into database with script 507
test driver program 209
testing 209
user-defined type 209
text files
delimiter 418
issues to consider before creating 418
procedure to read 419
Text property
effects on different controls 302
TextBox 291
multiline
selecting line of text by double-clicking 315
property
MultiLine 316
WrapContents 316
textfiles
reading and writing 418–420
TextWriter 417
the art of programming 4, 10
inspiration 10
money but no time 11
mood setting 11
time but no money 11
where not to start 10
your computer 11
thinking outside the box 190
this()
called from constructor 216
thread
execution context 384
thread context 384
thread queue 384
ThreadPool 382
ThreadPool class 399
number of default worker threads 399
starting threads with 400
threads 382–405
asynchronous method calls 400
BackgroundWorker class 396
BackgroundWorker events 396
blocking with Thread.Join() 392
blocking with Thread.Sleep() 391
creating managed threads 385
executing on single-processor system 384
foreground vs. background 394
ParameterizedThreadStart delegate 390
passing ThreadStart delegate to Thread constructor 389
preempted 384
running asynchronous methods with delegates 400
setting Thread.IsBackground property 394
starting managed threads 389
thread state 389
ThreadPool class 399
ThreadStart delegate 389
time-slicing 384
ThreadStart delegate 388
timeless way 688
time-slicing 384
TimeSpan
passing to Thread.Sleep() method 391
TimeSpan structure
example use of 310
title bar
window 293
ToolStripMenuItem
constructor usage 313
ToolStripMenuItems
declaring and creating 313
transitivity
exhibited by inheritance hierarchies 257
Transmission Control Protocol (TCP) 458
transport layer 458
tree command 36
try/catch statement 138
type 257
diagram 115
value type
behavior 116
type coercion 265
types 115–119
array 164
predefined 115
mapping to system namespace structures 118
reference 115
behavior 116
value 115
value range table 118

U

UDP 450, 458
UML 15, 190, 234, 250
class diagram 193
composite aggregation 237, 239
expressing abstract class 268
expressing inheritance 258
expressing interfaces 271
expressing realization 271
expression aggregation 236
realization
diagram
expanded form 272
simple form 272
sequence diagram
engine object creation 244
sequence diagrams 240, 241
simple aggregation 237
stereotype 194
using to tame conceptual complexity 234
UML class diagram
purpose of 193
UML design tool
Magic Draw 241
Unified Modeling Language (UML) 15
uniqueidentifier
use as primary key
example 504
unmanaged code 89
update command
SQL
commands
update 509
URI 450
URL 450
User Datagram Protocol (UDP) 458
user-defined types 191
using 114
using directive 111
utility methods
definition of 201

V

value objects 494, 495
spanning application layers 495
value parameters 219
value semantics 225
value types 115

© 2008 Rick Miller — All Rights Reserved

ValueType class 118
variable 47
 definition 116
verb phrases 46
verbatim string literals 412
verbs 46
vertical access 274, 617
view 695
virtual
 keyword to allow method overriding 266
Virtual Execution System 87
Virtual Execution System (VES) 86, 89
virtual machine 86
 and the common language infrastructure 86
virtual machines 86, 87
visible region
 of a window 293
Visual C# Express Edition
 building project 36
 creating project with 33
 creating projects with 32
 installing 32
 locating project executable file after build 36
void 114

W

well-behaved objects 614
WellKnownObjectMode.SingleCall mode 469
WellKnownObjectMode.Singleton mode 469, 470
whole object 235
whole objects 236
whole/part class relationship 235
Width property 302
window
 basic functionality provided by 293
 message categories 295
 message prefixes 295
 parts of 293
 title bar 293
 visible region 293
window application
 execution thread 294
window coordinates 297, 299
window coordinates diagram 298
window message routing 294
window messages
 trapping with IMessageFilter interface 296
window types
 dialog boxes 292
 floating 292
 multiple-document interface (MDI) 292
 standard 292
 tool 292
windows
 messages
  WM_CHAR 296
  WM_KEYDOWN 296
  WM_KEYUP 296
  WM_MOUSEMOVE 296
  WM_MOUSEWHEEL 296
windows events
 processing 293
windows executable
 compiler switch 293
 creating 293
Windows Task Manager
 using to show applications and processes 383
word 80, 81
world
 imperfect understanding of 668

X

XML documentation
 generating from command line 71
XML serialization 413
XMLSerializer 417
XMLSerializer class 416

© 2008 Rick Miller — All Rights Reserved

Master the Complexities of C# .NET And Object-Oriented Programming

C# For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming powers beyond ordinary introductory texts in its comprehensive coverage and audacious style. C# For Artists is the only book of its kind that succinctly addresses the art, philosophy, and science of C# .NET and object-oriented programming. Rick Miller presents material other authors shy away from or ignore completely. C# For Artists will help you smash through the barriers preventing you from mastering the complexities of object-oriented programming with C# and Microsoft's .NET Framework. Start taming complexity now! Read C# 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
C# 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 System.Windows.Forms components
Learn the secrets of thread programming to create multithreaded applications
Master the complexities of generic collections 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 objects that can be used predictably and reliably in C# .Net applications
Learn how to use MSBuild to manage large programming projects
Create multitiered database applications with the help of Microsoft's Enterprise Library
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 numerous tables, lots of pictures, and tons of code examples – 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...!

"A refreshing approach to a complex topic. Thank you!"

"The chapter on database access is worth the price alone..."

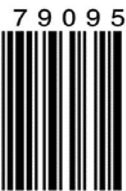
"Nicely done. Your books are different from all the rest."

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. Rick is the owner of Pulp Free Press and makes every effort to produce superior-quality programming textbooks. He's written and published two other titles: C++ For Artists and Java For Artists.

"Read C# For Artists Today – Energize your Programming Skills for Life!"

Pulp Free Press

ISBN 13 - 9781932504071
ISBN 1932504079



$79.95