Chapter 1

Three On A Beach

An Approach To The Art Of Programming

Learning Objectives

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

Introduction

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

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

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

The Difficulties You Will Encounter Learning Java

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

Required Skills

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

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

The Planets Will Come Into Alignment

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

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

How This Chapter Will Help You

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

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

Project Management

Three Software Development Roles

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

Analyst

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

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

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

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

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

Architect

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

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

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

Programmer

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

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

A Project-Approach Strategy

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

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

You Have Been Handed A Project – Now What?

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

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

Strategy Areas of Concern

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

Application Requirements

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

Problem Domain

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

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

Programming Language Features

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

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

High-Level Design & Implementation Strategy

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

Procedural-based Design Approach

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

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

Object-Oriented Design Approach

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

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

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

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

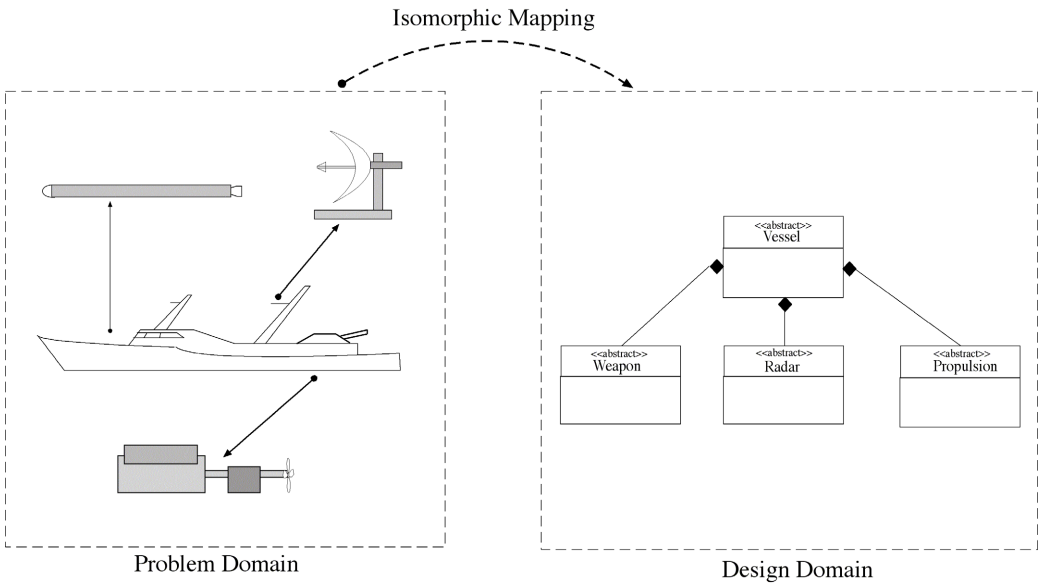


Figure 1-1: Isomorphic Mapping Between Problem Domain and Design Domain

Think Abstractly

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

The Strategy In A Nutshell

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

Applicability To The Real World

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

The Art of Programming

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

Don't Start At The Computer

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

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

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

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

Inspiration Strikes At The Weirdest Time

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

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

Own Your Own Computer

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

You Either Have Time and No Money, or Money and No Time

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

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

The Family Computer Is Not Going To Cut It!

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

Set The Mood

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

Location, Location, Location

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

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

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

Concept Of The Flow

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

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

The Stages of Flow

Like sleep, there are stages to the flow.

Getting Situated

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

Restlessness

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

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

Settling In

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

Calm and Complete Focus

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

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

Be Extreme

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

The Programming Cycle

Plan

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

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

Code

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

Test

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

Integrate/Test

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

Refactor

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

Repeat

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

The Programming Cycle Summarized

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

A Helpful Trick: Stubbing

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

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

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

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

Fix The First Compiler Error First

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

Managing Project Complexity

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

Conceptual Complexity

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

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

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

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

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

Managing Conceptual Complexity

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

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

The Unified Modeling Language (UML)

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

Physical Complexity

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

The Relationship Between Physical and Conceptual Complexity

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

Managing Physical Complexity

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

Maximize Cohesion – Minimize Coupling

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

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

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

Java Source File Structure

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

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

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

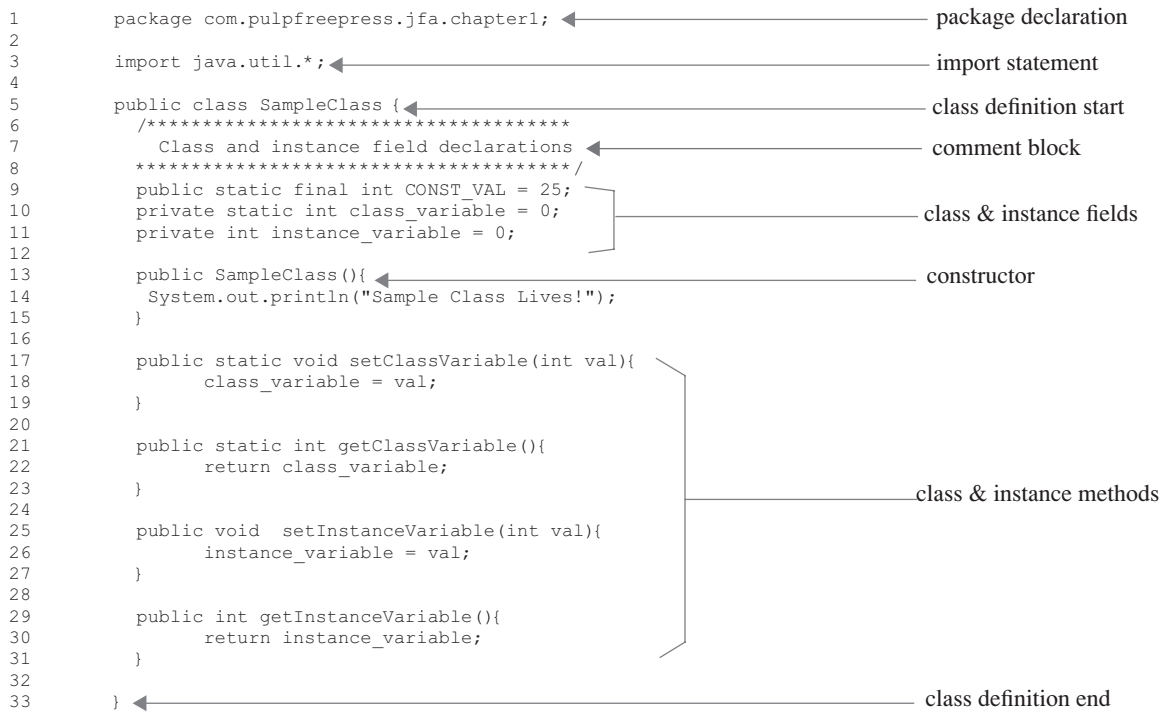
Line 3 is an import directive. An import directive does not physically import anything into the Java source file. Rather, it allows programmers to use shortcut names for components belonging to included package names.

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

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

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

1.1 SampleClass.java



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

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

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

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

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

SampleClass In Action

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

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

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

1.2 ApplicationClass.java

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

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

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

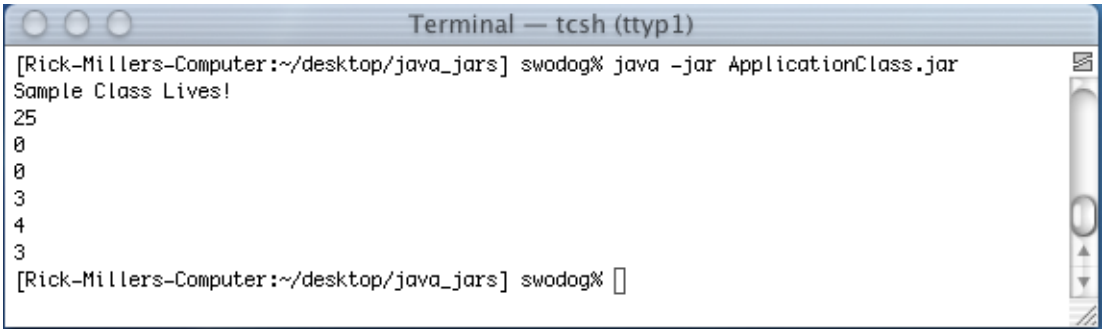


Figure 1-2: Results of Running Example 1.2

General Rules For Creating Java Source Files

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

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

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

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

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

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

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

Table 1-1: Java Source File Rules Summary

Rule-of-Thumb: One Class Per File

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

Avoid Anonymous, Nested, And Inner Classes

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

Create A Separate Main Application File

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

Packages

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

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

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

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

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

`com.pulpfreepress.jfa.chapter1`

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

`com.pulpfreepress.`*`package_name`*

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

`lastname.firstname.`*`project_name`*

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

`miller.rick.RobotRat`

Commenting

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

Single-Line Comments

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

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

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

Multi-Line Comments

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

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


Javadoc Comments

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

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

1.3 TestClass.java

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

Generating Javadoc Example Output

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

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

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

Identifier Naming - Writing Self-Commenting Code

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

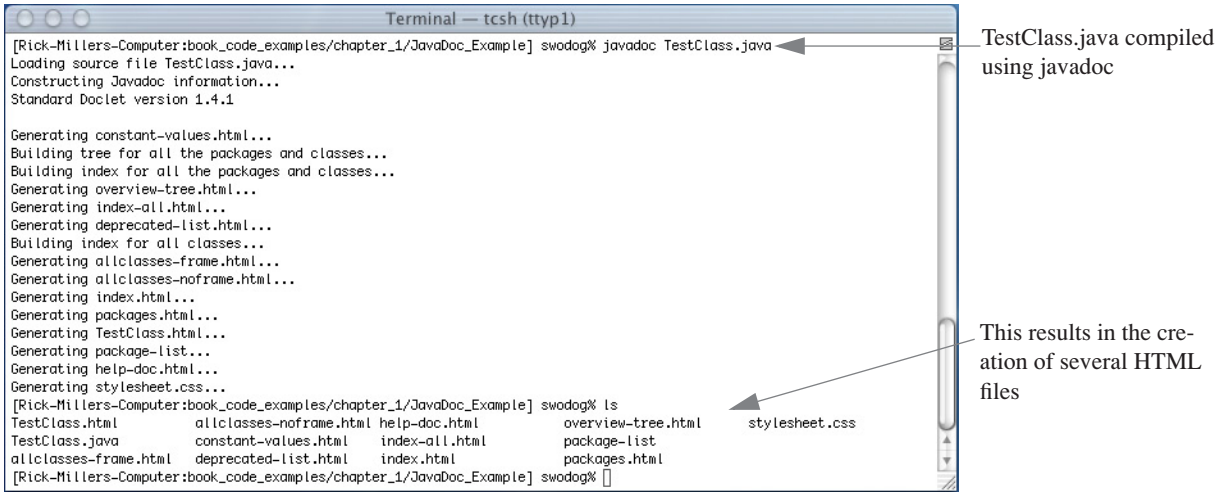


Figure 1-3: javadoc Tool Being Used to Generate TestClass API Documentation

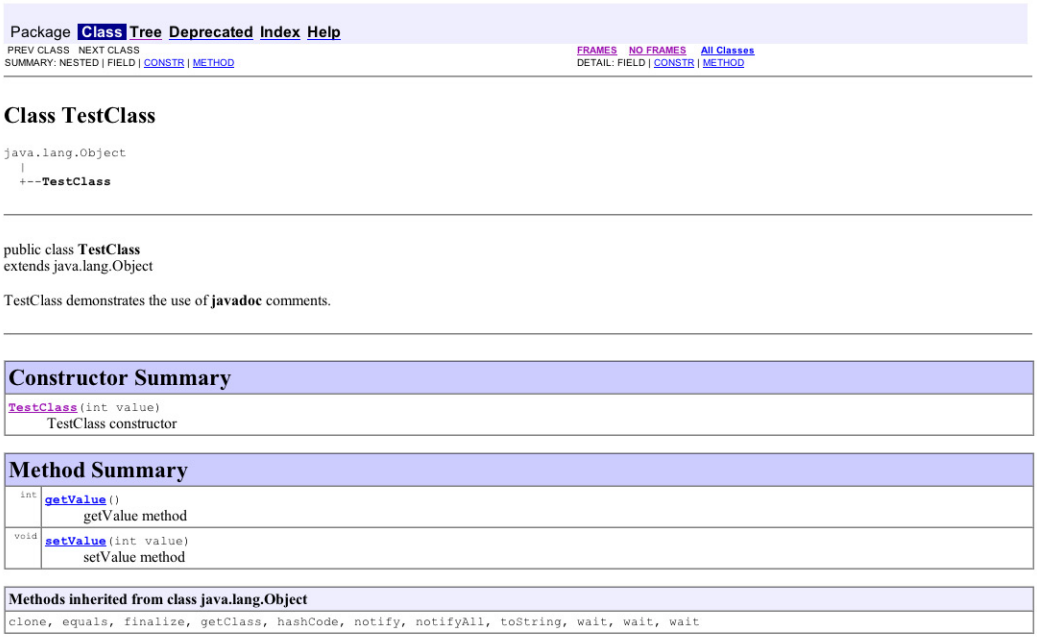


Figure 1-4: Example HTML Documentation Page Created With javadoc

Benefits of Self-Commenting Code

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

Coding Convention

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

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

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

Class Names

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

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

Table 1-2: Class Naming Examples

Constant Names

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

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

Table 1-3: Constant Naming Examples

Variable Names

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

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

Table 1-4: Variable Naming Examples

Method Names

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

Method Name	Comment
`printFloor`	Multiple-word method name. First letter of first word is lower-case; first letter of second word is upper-case. The first word is an action word.
`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 1-5: Method Naming Examples

Summary

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

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

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

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

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

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

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

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

Maximize cohesion — minimize coupling!

Skill-Building Exercises

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

each of the concepts listed below:

Number of oranges

Required waivers

Day of week

Month

People in line

Next person

Average age

Student grades

Final grade

Key word

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

Maximum student count

Minimum employee pay

Voltage level

Required pressure

Maximum array size

Minimum course load

Carriage return

Line feed

Minimum lines

Home directory

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

Sort employees by pay

List student grades

Clear screen

Run monthly report

Engage clutch

Check coolant temperature

Find file

Display course listings

Display menu

Start simulation

Suggested Projects

1. **Feng Shui:** If you haven't already done so, stake your claim to your own quiet, private space where you will work on your programming projects. If you are planning on using the school's programming lab, stop by and familiarize yourself with the surroundings.

2. **Procure and Install IDE:** If you are doing your programming on your own computer make sure you have procured and loaded an integrated development environment (IDE) that will meet your programming requirements. If in doubt, check with your instructor.

3. **Project-Approach Strategy Checklist:** Familiarize yourself with the project-approach strategy checklist in appendix A.

4. **Obtain Reference Books:** Seek your instructor's or a friend's recommendation of any Java reference books that might be helpful to you during this course. There are also many good computer book-review sites available on the Internet. Also, there are many excellent Java reference books listed in the reference section of each chapter in this book.

5. **Web Search:** Conduct a Web search for Java and object-oriented programming sites. Bookmark any site you feel might be helpful to you as you master the Java language.

Self-Test Questions

1. List at least seven skills you must master in your studies of the Java programming language.

2. What three development roles will you play as a student?

3. What is the purpose of the project-approach strategy?

4. List and describe the four areas of concern addressed in the project-approach strategy.

5. List and describe the five steps of the programming cycle.

6. What are the two types of complexity?

7. What is meant by the term *isomorphic mapping*?

8. Why do you think it would be helpful to write self-commenting source code?

9. What can you do in your source code to maximize cohesion?

10. What can you do in your source code to minimize coupling?

References

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

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

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

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

Notes

