

Team LIB

Team LIB



Ground-Up Java

by Philip Heller

ISBN:0782141900

Sybex © 2003 (488 pages)

In addition to learning the core Java language, you will also acquire a broad understanding of vital programming concepts, including variables, control, memory, indirection, compilation, and calling.

Table of Contents

[Ground-Up Java](#)

[Introduction](#)

[Chapter 1](#) - An Introduction to Computers That Will Actually Help You in Life

[Chapter 2](#) - Data

[Chapter 3](#) - Operations

[Chapter 4](#) - Methods

[Chapter 5](#) - Conditionals and Loops

[Chapter 6](#) - Arrays

[Chapter 7](#) - Introduction to Objects

[Chapter 8](#) - Inheritance

[Chapter 9](#) - Packages and Access

[Chapter 10](#) - Interfaces

[Chapter 11](#) - Exceptions

[Chapter 12](#) - The Core Java Packages and Classes

[Chapter 13](#) - File Input and Output

[Chapter 14](#) - Painting

[Chapter 15](#) - Components

[Chapter 16](#) - Events

[Chapter 17](#) - Final Project

[Appendix A](#) - Downloading and Installing Java

[Appendix B](#) - Solutions to the Exercises

[Glossary](#)

[Index](#)

[List of Figures](#)

[List of Tables](#)

Team LIB

Team LIB

Back Cover

This is the first effective Java book for true beginners. Sure, books before now focused on basic concepts and key techniques, and some even provided working examples on CD. Still, they lacked the power to transform someone with no programming experience into someone who sees, who really “gets it.”

Working with *Ground-Up Java*, you will definitely get it. This is due to the clarity of Phil Heller’s explanations, and the smoothly flowing organization of his instruction. He’s one of the best Java trainers around.

But what’s really revolutionary are his more than 30 animated illustrations. Each of these small programs, visual and interactive in nature, vividly demonstrates how its source code works. You can modify it in different ways, distinctly altering the behavior of the program. As you experiment with these tools—and you can play with them for hours—you’ll gain both the skills and the fundamental understanding needed to complete each chapter’s exercises, which steadily increase in sophistication. No other beginning Java book can take you so far, so quickly, and none will be half as much fun.

About the Author

Philip Heller is a consultant, author, educator, and novelist. He is the lead author for Sybex’s best selling *Java Certification Study Guide* and *Java Exam Notes* as well as a leading educator for Java University and a well-known speaker on Java topics. Phil helped create the Java programmer and developer exams for Sun and is their leading certification trainer. Phil is currently writing the second volume in the *Grandfather Dragon* series.

Ground-Up Java

Philip Heller

Associate Publisher: Joel Fugazzotto
Acquisitions Editor: Denise Santoro Lincoln, Tom Cirtin
Developmental Editor: Tom Cirtin
Production Editor: Dennis Fitzgerald
Technical Editor: Marcus Cuda
Copyeditor: Sean Medlock
Composer: Maureen Forsys, Happenstance Type-O-Rama
Graphic Illustrator: Jeffrey Wilson, Happenstance Type-O-Rama
CD Coordinator: Dan Mummert
CD Technician: Kevin Ly
Proofreaders: Emily Husan, Laurie O'Connell, Nancy Riddiough
Indexer: Ted Laux
Cover Designer/Illustrator: Richard Miller, Calyx Deisgns

Copyright © 2004 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 2003110719

ISBN: 0-7821-4190-0

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the United States and/or other countries.

Screen reproductions produced with FullShot 99. FullShot 99 © 1991–1999 Inbit Incorporated. All rights reserved. FullShot is a trademark of Inbit Incorporated.

The CD interface was created using Macromedia Director, COPYRIGHT 1994, 1997-1999 Macromedia Inc. For more information on Macromedia and Macromedia Director, visit <http://www.macromedia.com>.

TRADEMARKS: SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

*To Laura, on whose violin
Are played the songs of spheres and heroes,
Above this world's mortal din,
Above the plane of ones and zeroes.*

Acknowledgements

First and foremost gratitude to Denise Santoro Lincoln, Tom Cirtin, and Steve Cavin. Thanks to Michelle, Ricardo, and everyone at PB&G Productions for keeping me out dancing when I should have been writing. Thanks always to Simon Roberts, Suzanne Blackstock, and Kathy Collina. And thanks to all the aces at Sybex: Dennis Fitzgerald, Sean Medlock, Kevin Ly, Dan Mummert, and Maureen Forsys and Jeff Wilson at Happenstance Type-O-Rama.

Introduction

Overview

This book is unique. There's nothing like it. It is the first of its kind. It's important that you understand why, so please read on.

For a long time I thought it was impossible to write an introductory Java programming book that could be understood by people with no programming experience. It would be like a fish writing about water. No one has better knowledge of the subject matter, but it takes more than that to introduce a topic to a newcomer. Fish are intimately accustomed to water, and they can't relate to us land mammals, who need to have everything explained and broken down. A fish might say, "Wiggle your tail fin to swim forward, and don't forget to use your gills." That would be glaringly obvious to another fish, but useless to you and me. It's *hard* for a fish to imagine what life would be like without tail fins or gills. A book about water, even if the wisest fish in the ocean wrote it, would be full of accurate, but useless, information.

The same is true about Java. Programming is a craft, like playing a musical instrument or glassblowing. And like any other craft, it has its conventions, jargon, and techniques. For practitioners of the craft, those conventions, jargon, and techniques become deeply ingrained habits, household language, and the events of everyday life. It's very difficult to write about one's own "habitat."

In the 1970's, a language called C became popular. In the 1980's, C was modified to support object-oriented programming. The modified language was called C++. This is an example of craft jargon. In C, the symbol "+" means, very broadly speaking, "a bit more." So C++ means "C and a bit more," and the meaning is clear to any C programmer.

The 1990's saw another evolution. C++ is a highly effective language, but it can also be difficult. Moreover, it had no innate support for recently invented technologies, such as high-resolution multi-color displays, databases, or the World Wide Web. The new evolution was called Java. The name isn't a play on words and it isn't an abbreviation for anything. Java abandoned the parts of C++ that had proved to be more trouble than they were worth, and it added support for modern technologies. Sometimes people called it "C++-++". There's another symbol, "-", that roughly means "a bit less." So "C++-++" means "C++ and a bit less and then a bit more."

Java caught on like a midsummer bonfire. A huge portion of the C and C++ programming population switched at once to Java and never looked back. Why were so many programmers able to make the switch so easily? I was one of them. I had been earning a living programming in C++. I took a year off to write a novel about some dragons. I ran out of money before I finished the novel. Luckily, it was a month after Java was introduced. Within weeks I considered myself a competent Java programmer, and within months I was teaching it and writing about it. The credit goes not to me but to the designers of Java. If you know C and C++, Java is easy. It's like learning Portuguese if you already speak Spanish and Italian. Like everyone else who learned Java at that time, I had years of experience with the concepts, techniques, and jargon that was needed.

But what about people who don't have any programming experience?

When I was learning Java, there were two books on the subject. Today there are thousands. (I'm responsible for a few of them.) Not one of them, *except the one that you're holding right now*, does a good job of presenting programming concepts from the ground up. The others are accurate for the most part, but they aren't helpful.

So I had to ask myself: can I introduce Java from the ground up, concept by concept? Eventually I realized that I could only do it if I could use something more than words and pictures. Which brings me to why this book is unique. It is unique because ...

The Illustrations are Alive!

I realized that what I really wanted was a magic blackboard.

Think of a computer as a huge set of boxes, each box containing a number. The numbers represent text or colors or data, or whatever else can be modeled by a program. The numbers change over time in complicated ways. Describing the life cycle of a program is almost impossible if you can only use words and pictures. I wanted to create pictures that would change over time. And I wanted something beyond animated cartoons that would be the same each time you watched them. I wanted living illustrations that would respond to your curiosity. I wanted to give you the power to ask “what if ...” questions of the illustrations.

I wanted something that can only be done on a computer.

The CD-ROM that comes with this book has more than 30 *animated illustrations*. These are programs that you run on your computer. The book gives you complete instructions on how to use them. The illustration on the next page is an example.



This is a screenshot of NestedLoopLab, which appears in [Chapter 5, “Conditionals and Loops.”](#) The text in the upper-central part of the screen (“int color = 5” and so on) is Java code. The swirly image at the bottom is the result of running the code. The various controls let you vary the code, experimenting with different values until you get a feel for what the program is doing.

The animated illustrations are like training wheels on a bicycle. When you first learn to ride, there are so many things that can go wrong. Without training wheels you spend a lot of just time crashing and getting back up. Training wheels let you develop the right sense of balance. The animated illustrations won’t let you create code that crashes. They provide a safe environment in which you can develop the right sense of balance.

Later, of course, it’s time to take off the training wheels. At the end of each chapter you’ll find a set of exercises that will have you writing your own code. Suggested solutions to the exercises appear at the back of the book.

To the best of my knowledge, [Ground-Up Java](#) is the first book ever to use animated illustrations. So we have no data on how effective they are as a teaching tool. My guess is that they are worth their weight in gold. Everyone who has seen them has been very enthusiastic. But you are the most qualified judge. Try them! Please let me know what you think. You can e-mail your comments to groundupjava@sqsware.com. I’m especially interested in knowing which animated illustrations worked the best for you, and which ones didn’t. I’d also like to hear any suggestions you might have for more animations to appear in future revisions of this book. You are invited to be part of the development of animated illustrations as a new technology for learning.

And now...

Team LIB

← PREVIOUS

NEXT →

It's Time To Download and Install Java

Before you can start writing or running Java programs, you need to download some software. (The animated illustrations are Java programs, so they won't run if you don't do the download.)

Downloading is free. After Java is loaded on your hard drive, you have to follow a few steps to install it. These aren't difficult, but there's room for error, so please be careful. Complete instructions are explained in [Appendix A, "Downloading and Installing Java."](#)

And now you're ready. Have fun!

Team LIB

← PREVIOUS

NEXT →

Chapter 1: An Introduction to Computers That Will Actually Help You in Life

Overview

Java is a programming language that tells computers what to do. This chapter will look at what computers really are, what they can do, and how we use programming languages to control them.

We will begin by exploding the common myth that computers deal only with 0s and 1s. Once we establish what computers really process, we will look at the kind of processing they perform.

This is emphatically *not* an intellectual exercise. Spending a bit of effort here will make your life much easier in the chapters that follow. Many concepts that appear later in this book, such as data typing, referencing, and virtual machines, will make very little sense unless you understand the underlying structure of computers. Without this understanding, learning to program can be confusing and overwhelming. With the right fundamentals, though, it can be enjoyable and stimulating.

Memory: Not Exactly 0s and 1s

No doubt you've heard that computers only process 0s and 1s. This can't possibly be true. Computers are used to count votes in elections, so they must be capable of counting past 1. Computers are also used to model the behavior of subatomic particles whose masses are tiny fractions, so they must be capable of processing fractions as well as whole numbers. They're used for writing documents, so they must be capable of processing text as well as numbers.

On the most fundamental level, computers do not process 0s and 1s, or whole numbers, or fractions, or text. Computers are electronic circuits, so all they really process is electricity. Computer components are designed so that their internal voltages are either approximately zero or approximately 5 or 6 volts. When part of a computer circuit carries a voltage of 5 or 6 volts, we say that it has a value of 1. When part of a circuit carries zero voltage, we say that it has a value of 0. (Fortunately, this is all the electronics knowledge you need to become a master programmer.)

It's all a matter of interpretation. Voltages are interpreted as 0s and 1s. As you'll see later in this chapter and in [Chapter 2, "Data,"](#) the 0s and 1s are organized into clusters that are interpreted as numbers. More sophisticated parts of the computer interpret those numbers as codes that represent fractions, or text, or colors, or images, or any of the other myriad classes of objects that can be represented in a computer.

A modern computer contains billions of microscopic components, each of which has a value of 0 or 1. Any circuit where we only care about the approximate values of the voltages is known as a *digital circuit*. Computers that are made of digital circuitry are known as *digital computers*.

Note The opposite of digital is *analog*. In an analog circuit, we care about the exact voltages of the components. Analog circuits are ideal for certain applications, such as radios and microwave ovens, but they don't work so well for computers. Analog computers were used in the 1940s, but they were an evolutionary dead end. All modern computers are digital.

One simple but useful type of digital circuit is known as *memory*. A memory circuit just stores a digital value (0 or 1, because we programmers don't have to think about voltages). A single unit of memory is called a *bit*, which is an abbreviation for "binary digit." You can think of a bit as a microscopic box, the contents of which are available to the rest of the computer. From time to time the computer might change the contents. Bits are usually drawn as shown in [Figure 1.1](#).



Figure 1.1: A bit

Bits are usually organized in groups of eight, known as *bytes*. [Figure 1.2](#) shows a byte that contains an arbitrary combination of 0s and 1s.

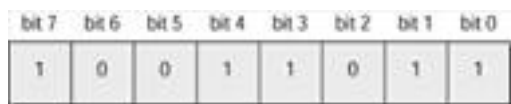


Figure 1.2: A byte

Note that the individual bits are numbered from right to left, and that the numbering starts from 0. Computer designers always start numbering things from 0 rather than 1. This is true whether they are numbering bits in a byte, bytes in memory (as we are about to see), or components in an array (as we will see in [Chapter 6](#)).

A byte can contain 256 combinations of bit values: 2 possibilities for bit #0 times 2 possibilities for bit #1 times 2 possibilities for bit #3, and so on up through bit #7.

If you looked at a computer through a microscope and saw the byte shown in [Figure 1.2](#), you might wonder what value it contained. You would see the 0s and 1s, but what would they mean? It's a great question that has no good answer. A byte might represent an integral number, a fraction, part of an integer or fraction, a character in a document, a color in a picture, or an instruction in a program. It all depends on the byte's context. As a programmer, you are the one who dictates how each byte will be interpreted.

Memory Organization

Typically, a modern personal computer contains several hundred million bytes of memory. The prefix *mega* (abbreviated *M*) means million, so we could also say that a computer has several hundred megabytes or *MB*. Programs and programmers need a way to distinguish one byte from another. This is done by assigning to each byte a unique number, known as the byte's *address*. Addresses begin at 0. [Figure 1.3](#) shows 4 bytes.

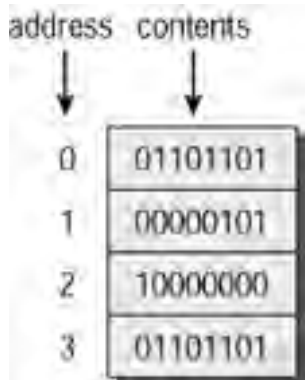


Figure 1.3: Several bytes

If [Figure 1.3](#) showed 512 MB and was drawn to the same scale, it would be about 2,000 miles high.

A single byte is not very versatile, because its value is limited to 256 possibilities. It doesn't matter whether the byte represents a number or a letter or anything else—in computer applications, 256 of *anything* isn't much of a range. For this reason, computers often use groups of bytes. Two bytes, taken together as a unit, can take on 256 times 256 possible values, or 65,536. Four bytes can take on 256 times 256 times 256 times 256 values, or 4,294,967,296. This is where it starts to be useful. Eight bytes can take on approximately 20 quintillion different values.

Memory is usually used in chunks of 1, 2, 4, or 8 bytes. (Later we will see that arrays and objects use chunks of arbitrary size.) The chunks can represent integral numbers, fractions, text, or any other kind of information. From this perspective, we can see that the statement "Computers only deal with 0s and 1s" is true only in a very limited sense.

Think of it this way: A computer is a digital circuit, and we think of its components as having values that represent 0s or 1s. But if we look one level below the digital components, we see only electricity, not numbers. And if we look one level above the digital components, we see that the bits are organized into chunks of 1 or more bytes that represent many types of information.

In addition to various types of data, memory can also store the instructions that operate on data. In the [next section](#), we will look at a very simple computer and see how instructions and data interact.

A Very Simple Computer

This chapter will introduce a very simple computer called SimCom. SimCom is imaginary. Or, to use a more respectable term, it is *virtual*. Nobody has ever built a SimCom, but it is simulated in one of the animated illustrations on the CD-ROM.

The processors that power your own computer, the Pentiums, SPARCs, and so on, are not very different qualitatively from SimCom. *Quantitatively*, however, there is a huge difference: the real processors have vastly more instructions, speed, and memory. SimCom is as simple as a computer can be while still being a useful teaching tool.

The point of this section is not to make you a master SimCom programmer. The point is to use SimCom to introduce certain principles of programming. Later in this book, the same principles will be presented in the context of Java. These principles include

- High-level languages
- Loops
- Referencing
- Two's complement
- Virtual machines

In this section, you will see some typical processor elements that are quite low-level. Modern programming languages like Java deliberately isolate you from having to control these elements. However, it is extremely valuable to know that they exist and what they do on your behalf.

The architecture of SimCom is very simple. There is a bank of 32 bytes of memory; each byte can be used as an instruction or as data. There is one extra byte, called the *register*, which is used like scratch paper. Another component, called the *program counter*, keeps track of which instruction is about to be executed. [Figure 1.4](#) shows the architecture of SimCom.

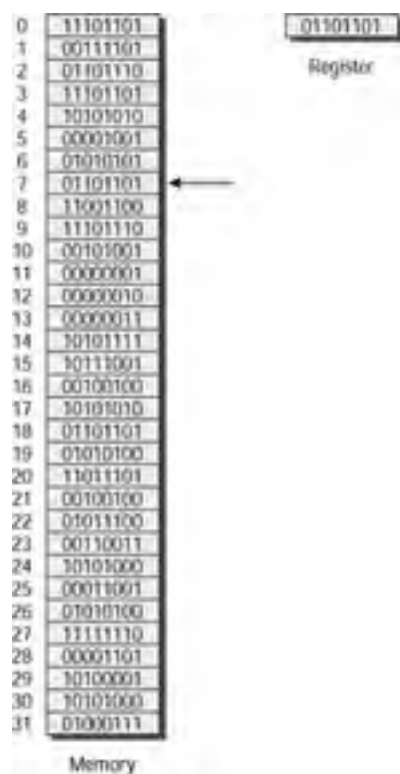


Figure 1.4: SimCom architecture

The arrow in the figure indicates the program counter. The next instruction to be executed will be byte #7. Note that byte addresses start at 0.

When SimCom starts up, it sets the program counter to 0. It then *executes* byte 0. (We'll see what this means in a moment.) Execution may change the register or a byte of memory, and it almost always changes the program counter. Then the whole process repeats: The instruction indicated by the program counter is executed, and the program counter is modified. This continues until SimCom is instructed to halt.

Bits 7, 6, and 5 of an instruction byte tell SimCom what to do. They are known as the *operation code* or *opcode* bits. Bits 4 through 0 contain additional instructions; they are called the *argument* bits. This division of bits is shown in [Figure 1.5](#).

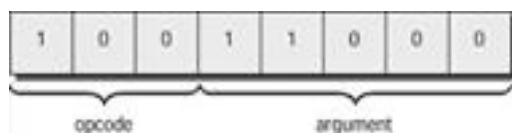


Figure 1.5: Opcode and argument bits

The SimCom computer has 7 opcodes. They are shown in [Table 1.1](#).

Table 1.1: Opcodes

Opcode	Function	Abbreviation
000	Load	LOAD
001	Store	STORE
010	Add	ADD
011	Subtract	SUB
100	Jump to different current instruction	JUMP
101	Jump if register is zero	JUMPZ
110 or 111	Halt	HALT

The 5 argument bits contain a value that is the base-2 address of a memory byte. The LOAD opcode copies the contents of this address into the register. For example, suppose the current instruction is 0000011. The opcode is 000 (LOAD), and the argument is 00011 (which is the base-2 notation for 3). When the instruction is executed, the value in byte #3 is copied into the register. Note that the value 3 is *not* copied into the register. The argument is never used directly; it is always an address whose contents are used.

The STORE opcode copies the contents of the register in the memory byte whose address appears in the argument. For example, 00100001 causes the register to be copied into byte #1.

The ADD opcode adds two values. One value is the value stored in the byte whose address appears in the argument. The other value is the contents of the register. The result of the addition is stored in the register. For example, suppose the register contains 00001100, and byte #1 contains 00000011. The instruction 01000001 causes the contents of byte #1 to be added to the contents of the register, with the result being stored back in the register. Note that the argument (00001) is used *indirectly*, as an address. The value 00001 is not added to the register; rather, 00001 is the address of the byte that gets added to the register.

The SUB opcode is like ADD, except that the value addressed by the argument is subtracted from the register. The result is stored in the register.

After each of these four opcodes is executed, the program counter is incremented by 1. Thus, control flows sequentially through memory. The remaining three opcodes alter this normal flow of control. The JUMP opcode does not change the register or memory; it just stores its argument in the program counter. For example, after executing 10000101, the next instruction to be executed will be the one at byte 00101, which is the base-2 notation for 5.

The JUMPZ opcode inspects the register. If the register contains 00000000, the program counter is set to the instruction's argument. Otherwise, the program counter is just *incremented* (that is, increased by 1) and control flows normally. This is a very powerful opcode, because it enables the computer to be sensitive to its data and to react differently to different conditions.

Finally, the HALT opcode causes the computer to stop processing.

Let's look at a short program:

```
00000100
01000100
00100100
11000000
```

The first thing to notice about this program is that it's hard to read. Let's translate it to a friendlier format:

```
LOAD    4
ADD     4
STORE   4
HALT
```

The program doubles the value in byte #4. It does this by copying the value into the register, then adding the same value into the register, and then storing the result back in byte #4.

This example shows that anything is better than programming by manipulating 0s and 1s. These spelled-out opcodes and base-10 numbers are a compromise between the binary language of computers and the highly structured and nuanced language of humans. The LOAD 4 notation is known as *assembly language*. In assembly language, a line of code typically corresponds to a single computer instruction, and the programmer must always be aware of the computer's architecture and state. An *assembler* is a program that translates assembly language into binary notation.

Playing with SimCom

Unfortunately we couldn't package a SimCom with every copy of this book, but we have done the next best thing. The first animated illustration on the book's CD is a simulation of a SimCom in action.

Note If you don't already have Java installed on your computer, now is the time. If you're not sure how, please refer to [Appendix A, "Downloading and Installing Java."](#) which walks you through the entire process. Throughout this book you

will be invited to run an animated illustration program, and you will be given a command to type into your machine. It will all make sense after you go through [Appendix A](#).

To run the SimCom simulation, type the following at your command prompt:

```
java simcom.SimComFrame
```

The simulation allows you to load and run preexisting programs or create your own programs. [Figure 1.6](#) shows the simulation in action.

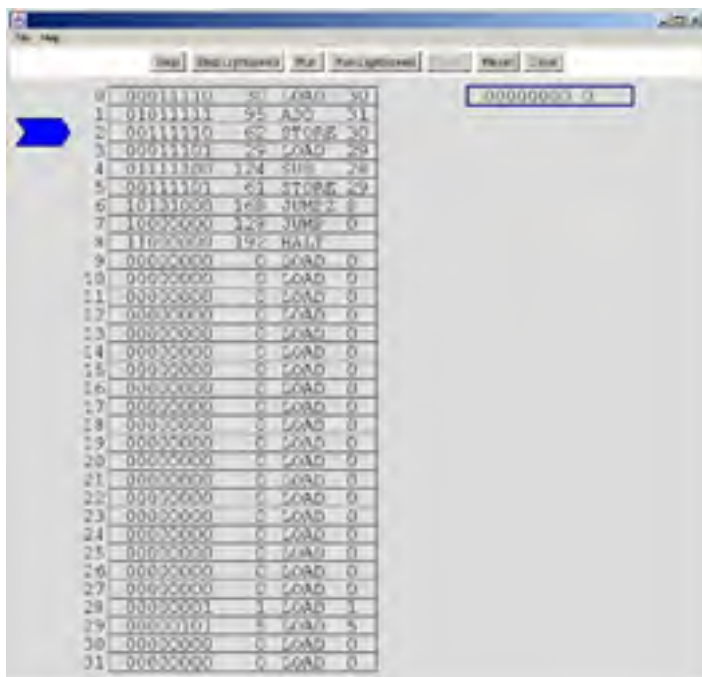


Figure 1.6: SimCom in action

Each byte of memory is displayed in three formats: base-2, base-10, and opcode-plus-argument. The register is only displayed in base-2 and base-10; since the register is never executed, there is no value in displaying which instruction *would* be executed. You can change any byte in memory by first clicking inside that byte. This will highlight and select the byte for editing. Then, if you click on the base-10 region, you will get a panel that lets you select a new base-10 value. If you click on the opcode region, you will get a panel that lets you select a new opcode. To change the argument, first click on the argument region of the selected byte. As you move the mouse, the closest byte address will light up. When the address you want is highlighted, click on it to set it as the argument.

Try executing a very simple program. Click File, Scenarios in the File menu, and select Load/Add/Store/. This program adds bytes 10 and 11 (not the numbers 10 and 11, but the contents of the memory bytes whose addresses are 10 and 11), and stores the result in byte 12. Initially, bytes 10 and 11 both contain zero, so to see interesting results you will have to change their values. To see the program in action, click the Step button. This executes the current instruction in slow motion. To run continuously, click the Run button, which plays the animation until a HALT instruction is executed. If you get tired of the slow motion, you can click Step Lightspeed or Run Lightspeed to get instant results. The Reset button reinitializes memory and sets the program counter to zero.

Try storing relatively large values in bytes 10 and 11. The largest value a byte can store is 255. What happens if you try to add 5 + 255?

Change the program so that byte 11 is subtracted from byte 10. What happens if byte 10 contains 5 and byte 11 contains 6?

When you are ready for a more interesting program, click Scenarios, Times 5 in the File menu. This program multiplies the contents of byte 31 by 5 and stores the result in byte 30. Experiment with a few values in byte 31 to convince yourself that it works. Remember to click the Reset button after each run.

This program might seem needlessly complicated. It's too bad the SimCom instruction set doesn't include a multiply opcode, but since it doesn't, wouldn't the following program be more straightforward?

```
LOAD    31
ADD     31
ADD     31
ADD     31
ADD     31
STORE  30
HALT
```

This is definitely more straightforward, but it is also less flexible than the version SimCom uses. That version uses a *loop*, a powerful construct that appears in all programming languages. Note that initially, byte 29 contains 5; this byte is a *loop counter* that controls how many times the loop will be executed. Lines 0 through 3 add whatever is in byte 31 (the value to be quintupled) to whatever is in byte 30 (the accumulating result). Then lines 3 through 5 subtract 1 from the loop counter. If the loop counter reaches zero, line 6 causes a jump to a HALT instruction. If the decremented loop counter has not yet reached zero, line 7 causes a jump back to line 0, which is the beginning of the loop.

Reset the Times 5 program. Change the value in byte 29 (the loop counter) from 5 to 6. Put a reasonable value in byte 31 and run the program. Notice that the program now multiplies

by 6. This is to be expected, because the value in byte 31 has been added one extra time to the accumulated result.

Now you can see how the looping version is more flexible than the repeated-addition version shown earlier. To modify the looping version so that it multiplies by 10 instead of 5, you just have to change the loop counter in byte 29. In the repeated-addition version, you have to make sure you add the right number of ADD 31 lines, and then make sure the STORE 30 and HALT lines are intact. That may not seem unreasonable to you, but what if you want the program to multiply by 30? With the looping version, you just change the loop counter. With the repeated-addition version, you will run out of memory.

As you experiment with the SimCom simulation, you will probably notice a few things:

- Specifying an instruction by selecting an opcode and an argument is much easier than figuring out what the base-10 value should be.
- Even so, SimCom programming isn't very easy.
- When you look at any byte, you can't tell if it is supposed to be an instruction or a value. For example, a byte that contains 100 might mean one hundred, or it might mean SUB 4.

The first two points suggest the need for higher-level programming languages. Hopefully, such languages will support sophisticated operations like multiplication and looping.

The Lessons of SimCom

The point of presenting SimCom in this chapter was to expose you to certain basic functions of programming. Those were high-level languages, loops, referencing, two's complement, and virtual machines. Now that you've been exposed, we can look at how SimCom supports those functions.

Programming with opcodes and arguments is certainly easier than specifying base-10 or (worse yet) base-2 values. But SimCom still forces you to think on the microscopic level. In the Times5 program, you have to remember that byte 29 is the loop counter and byte 30 is the accumulated result. You always have to remember what's going on in the register. High-level languages like Java isolate you from the details of the computer you're programming. (That probably sounds like a good thing, now that you have suffered through SimCom.)

Loops are basic to all programming. Computers are designed to perform repetitive tasks on large data sets, such as printing a paycheck for each employee, displaying each character of a document, or rendering each pixel of a scanned photograph. Loops are difficult to create on SimCom, because everything is hard on SimCom. Java uses simple and powerful looping constructs.

We will cover referencing much later in this book, in [Chapter 6, "Arrays."](#) For now, you've had a preview. Remember how SimCom never directly operated with an instruction's argument? The argument was always used as the address of the value to be loaded, added, etc. Now you should be used to the difference between the *address* of a byte and the *value* in that byte. When you program in Java, you don't have to worry about the address of your data, but you still have to think about its location. This will make more sense later on. For now, it's enough to understand the distinction between the value of data and the location of data.

Two's complement is a convention for storing negative numbers. On its surface, SimCom seems to deal only with positive numbers (and zero, of course). But subtraction is supported, and subtraction can lead to negative numbers. If you did the exercise where you modified the LoadAddStore program to make it subtract, you noticed that SimCom thinks 5 minus 6 equals 255. In a way, this is actually correct.

SimCom does not really exist. When you run the animated illustration, there is no actual SimCom computer doing the processing. The program simulates the computer's activity. Thus, SimCom is an imaginary processor that produces real results. As stated earlier in this chapter, an imaginary computer that is simulated on a real one is known as a *virtual computer*. You might have heard of the *JVM*, or *Java Virtual Machine*. Java programs, like SimCom programs, run on a virtual computer.

There is a powerful benefit to this arrangement. When you buy software for your personal computer, you have to check the side of the box to make sure the product works on your platform. If you own a Windows PC, it is useless to buy Macintosh software, just as it is useless to buy SPARC software for a Mac. This is because different manufacturers use different kinds of processors. The binary opcode for addition on one processor type might mean subtract to another type, and might be meaningless to a third type. Thus, software vendors have needed to create a different product for each computer platform they want to support.

Virtual computers do not have this limitation. No matter what kind of computer you're using, SimCom loads when it executes 000, stores when it executes 001, and multiplies by 5 when it executes the Times5 program.

The Java Virtual Machine is much more complicated than SimCom, but the same principle applies. Any Java program will run the same on any hardware. Of course, the JVM itself varies from processor to processor. This is why you had to specify your platform when you downloaded Java. From the JVM's point of view, your platform is known as the *underlying hardware*.

Exercises

Note Every chapter in this book ends with exercises that test your understanding of the material and make you think about issues raised in later chapters. The solutions are in [Appendix B](#).

1. A cluster of eight bytes can take on approximately 20 quintillion different values. (One quintillion is a 1 followed by 18 zeroes, or 10 to the 18th power.) Estimate the number of different values that a cluster of 16 bytes can have. Just estimate, do not count. Can you think of anything that comes in such quantities?
2. The SimCom animated illustration is written in Java. When you run the program, how many virtual machines are at work?
3. Write a SimCom program that adds 255 to the value in byte 31 and stores the result in byte 30. Observe the program's behavior. What do you notice?
4. Write a SimCom program that computes the square of the value in byte 31 and stores the result in byte 30. What happens when you try to compute the square of 254?
5. What features could be added to SimCom to make it more useful?

Chapter 2: Data

Overview

Computers process *data*—factual information, such as numbers, text, images, and sound—in a form that can be processed by electronic devices. That is the whole idea of computers. In this chapter, you will see how Java handles data. This chapter will cover the two most important things a program does with data:

- Declaring
- Assigning

Declaring and assigning are activities that we perform in the context of a compiled language such as Java. This chapter will begin by explaining what a compiled language really is. If you are already familiar with this topic, feel free to skip to the [next section, "Data Types."](#)

In the [previous chapter](#), we looked at the SimCom virtual machine and experienced its benefits and drawbacks. SimCom was not much of a computer, but it was valuable as a learning tool. The drawbacks mostly had to do with scale: SimCom did not have enough memory or commands to do anything very interesting. In this chapter, we leave SimCom behind and discuss Java itself.

Note Jumping into Java can be difficult if you're learning programming from the ground up. Even the simplest possible Java program uses many unfamiliar constructs, including classes, methods, arrays, access, and static code. We can't expect you to learn all these concepts before you look at a Java program. So beginning in this chapter, you will be asked to accept that certain parts of all Java programs have to be in place in order for the program to work at all. Eventually, later chapters will present everything you are being asked to accept.

Understanding Compiled Languages

The SimCom virtual computer is difficult to program. You have two options for specifying an instruction: You can enter a byte value, or you can specify an opcode and an address. You've probably found that specifying an opcode and an address is a much better approach, but it's still not very intuitive.

The language of programming with opcodes is known as *assembly language*. Every line of an assembly language program roughly corresponds to a computer instruction in memory. Real computers have much more memory than SimCom, and assembly programs that make real machines do something useful can be quite long. Such programs are created using a text editor. The resulting file is known as *source code*, and it must be translated into the appropriate binary values before it can be executed by a computer.

Conceivably, this translation could be done by people. In fact, in the very earliest days of programming, that's how it was done. However, computers can do a much better job of it. Any program that translates assembly code into computer base-2 code is called an *assembler*. [Figure 2.1](#) shows the flow from assembly language source code to executable computer code.

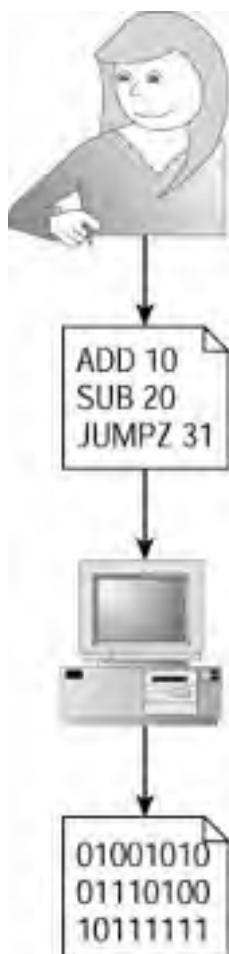


Figure 2.1: Assembly language

After a program has been assembled, it must be loaded into memory and the computer must be told to execute it. This is the job of the operating system.

Assembly programming has many shortcomings, all of which result from being too close to the underlying architecture. When you are forced to think in terms of the interrelationships among hardware components, it is difficult to also consider the domain of the problem you are trying to solve. For example, if you want to write a program to model weather patterns, you will be better off thinking about air currents and water vapor, not about opcodes and registers. To do that, you need a compiled language.

Compiled vs. Assembly Languages

A *compiled language* is like assembly language in the sense that a source program is created using a text editor, and the source must be translated into computer binary. The difference is that, unlike assembly code, a line of source code generally does *not* correspond to a single instruction. In fact, one great benefit of compiled languages is that you don't need to know anything at all about the underlying hardware.

[Figure 2.2](#) shows the flow from compiled language source code to executable computer code.

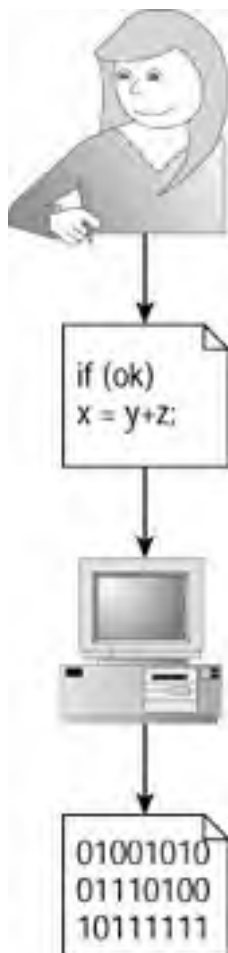


Figure 2.2: Compiled language

Each type of computer (Pentium or SPARC, for example) has its own instruction set and architecture, and hence its own assembly language. However, a compiled language can run on any target machine, provided there's a compiler that can translate it into the target machine's instruction set. For example, there are compilers that translate C++ into Pentium code, and other C++ compilers that produce code for SPARC processors.

Software can be developed much more efficiently with a compiled language than with assembly language. Moreover, in theory a company only needs to develop one version of a software product. When the product is finished, one compiler can be used to produce PC code, another compiler can be used to produce Macintosh code, and so on.

That's the theory. In practice it doesn't work so well. Certainly compiled languages are phenomenally more efficient for development than assembly languages. However, the ideal of developing once and compiling many times is just an ideal. There are differences among target computers that should be negligible, but are in fact significant. Source code that runs flawlessly on one platform may require considerable tweaking to run on a different platform. Multiple versions of source code have to be maintained. The process can get extremely expensive.

The Java Virtual Machine

Java is an *interpreted compiled language*. This means the compiler does not generate code that is specific to any particular processor. Instead, the compiler generates code for an imaginary processor: a *virtual machine*. The compiler does almost all the work. It checks for grammatical correctness, analyzes the structure of the source code, and breaks the source down into elementary units. It does everything except create code that can be run by a computer that exists in the physical world. The Java compiler's output is called *bytecode*, which is the binary format that is understood by the *Java Virtual Machine*, or *JVM*.

The JVM is a program that executes bytecode instructions. Like SimCom in [Chapter 1](#), the JVM's architecture is usually implemented in software rather than being built from circuit components. The JVM itself runs on physical hardware, so there is one version for Windows platforms, one for SPARC platforms, one for Mac platforms, and so on.

When you run a Java application, you are really running the JVM, which in turn loads and executes the bytecode for your application. All JVMs for all platforms execute bytecode in the same way. This means that with Java, you do not have to maintain different versions of source code for different platforms. One of the Java slogans is, "Write once, run anywhere." And it works. With Java, a program has exactly one version of source code. The result of compiling the source—the bytecode—will run on any platform for which a JVM is available.

[Figure 2.3](#) shows the evolution of a Java application from source code through execution.

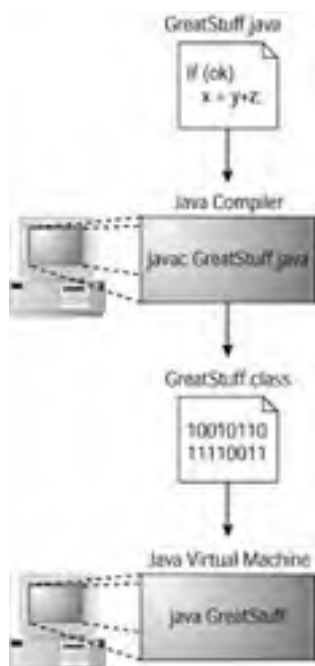


Figure 2.3: Evolution of a Java application

In [Figure 2.3](#), the source code is the file `GreatStuff.java`. All Java source files have to end with `.java` or the compiler won't touch them. The compiler produces one or more files of bytecode output. The bytecode files, also known as *class files*, always end with `.class`. To run a Java program, you type `java classname`, where `classname` is the name of the class file that contains the starting point of the program. Note that here you omit the `.class` suffix. `java` is the name of the JVM program which will read and execute the bytecode class file.

Now that you've seen how the Java compiler and Virtual Machine fit into the big picture, it's time to get acquainted with a fundamental concept of Java programming: data types.

Data Types

Imagine what would happen if SimCom accidentally treated bytes of data as if they were instructions, or instructions as if they were data. In the first case, the virtual machine would execute a random series of opcodes, producing nothing of value. In the second case, instructions would likely be modified (added to or subtracted from one another), again producing nothing of value.

The point is that SimCom uses memory for two different purposes, instructions and data, and each memory type must be treated appropriately. There are no facilities built into SimCom to guarantee appropriate treatment. You just have to be a careful programmer.

This distinction between memory uses is also found in Java and all other high-level languages. Fortunately, Java makes it impossible to execute data or do arithmetic on opcodes.

SimCom has no facilities for dealing with fractions, characters, or very large numbers, and negative numbers are mysterious. Java supports all these different characteristics of numbers. It does this by providing different *data types*. For now, you can think of a data type as a way of using memory to represent data. SimCom uses an eight-bit base-2 representation. Java provides several base-2 representations: two representations for numbers that might contain fractions, one for characters, and one for logical (true/false) values.

Processing a Java data type as if it were a different type would produce worthless results. Java protects you from this kind of problem by requiring you to declare all data types; the compiler enforces the integrity of your declarations. Of course, this will make much more sense later in this chapter, after we discuss declarations. Right now, let's look at Java's data types. Later on, you'll see how they're used.

Integer Data Types

In the terminology of programming, an *integer* is a data type that represents non-fractional numbers. In Java, all integer types are *signed*, meaning that both positive and negative values are supported (as is zero). Java's four integer types are shown in [Table 2.1](#).

Table 2.1: Java's Integer Data Types

Name	Size	Minimum Value	Maximum Value
byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807

Each data type shown in [Table 2.1](#) has a finite range. Wider ranges are accommodated by data types that require more memory. No type is unlimited – each has a minimum and a maximum value – but it is difficult to imagine exhausting the capacity of the `long` type, which ranges from minus nine quintillion to plus nine quintillion.

Java uses a format known as *two's complement* to represent negative numbers. In nearly all cases, the details of this representation are hidden from programmers, so you can go for a long time without having to know about it. However, there are times when a program will produce baffling results if you don't know about two's complement. Also, there are some arithmetic operators (discussed in [Chapter 3, "Operations"](#)) that only make sense if you know how negative numbers are represented.

Two's complement is an evolution of the classical base-2 notation that we all learned in elementary school. If you need a review, you can run the Simple Base 2 animated illustration on the CD-ROM. First run the Java setup script you created in [Appendix A](#) (assuming you haven't run it already), and type `java twoscomp.SimpleBase2Lab`. You see a ten-bit number. You can click on individual bits to change their values. When you're ready, click the Run button to see which number is represented. [Figure 2.4](#) shows SimpleBase2Lab in action.

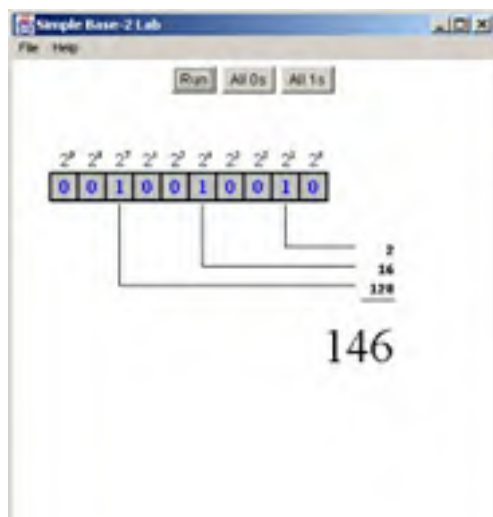




Figure 2.4: SimpleBase2Lab

Straightforward base-2 notation, as shown in the Simple Base 2 animation, is not exactly what computers use to represent numbers. Two's complement is more sophisticated than regular base-2, because of the way negative numbers are represented.

Imagine a car with an odometer that uses base-2 rather than base-10. There are a lot more digits than usual, and they roll over more frequently, but otherwise this odometer is like an ordinary odometer. Every time you drive another mile, the displayed number increases by 1. When the display is showing all 1s, and you drive one more mile, the odometer rolls over and shows all 0s. Thus if you wanted to get imaginative, you could say that in a way a display of all 1s represented -1 mile, because when you add one more mile, you get zero miles.

What about a display that consists of all 1s except for the rightmost digit, which is zero? (This would be 11111110 on an 8-bit odometer.) You could make a case that this reading represents -2 miles, because when you drive two more miles you get zero miles.

Here is another way to make the same case: if you were willing to break the law, you could open the odometer and roll it back manually. If it initially showed one mile and you rolled it back once, it would show zero miles. If you then rolled it back once more, it would show 11111111.

Figure 2.5 shows a base-2 odometer.

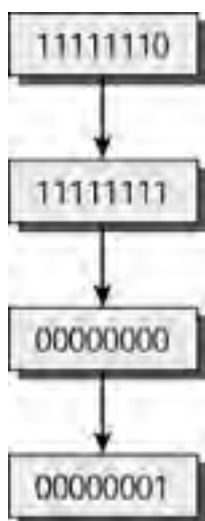


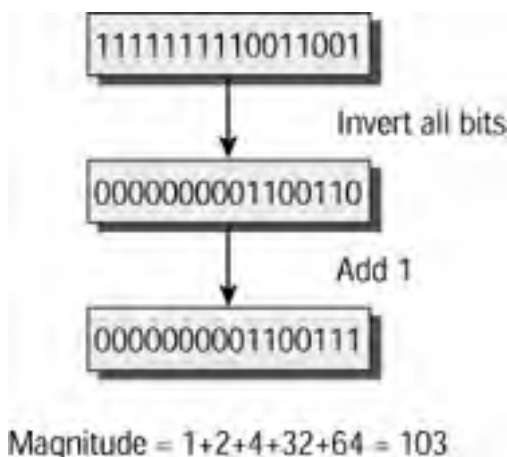
Figure 2.5: A base-2 odometer

Two's complement works like an odometer. A value of all 1s represents -1. Other values are assigned to ensure consistency. For example, with an 8-bit byte, a value of 11111110 represents -2. This makes sense, because adding 1 produces the "all 1s" representation for -1.

The general rules for two's complement are as follows:

- A value of all 0s represents zero.
- If the leftmost bit is 0, the number is positive. The remaining bits represent the value in base-2.
- If the leftmost bit is 1, the number is negative. To compute the magnitude of the value, invert all bits (changing 0s to 1s and 1s to 0s) and then add 1.

Figure 2.6 shows how to compute the value of the 16-bit short 1111111110011001.



Java uses a 16-bit data type called `char` to represent text characters. The data type can accommodate 2^{16} or 65,536 bit combinations, so 65,536 characters can be represented. This is more than enough to encode all European-based languages, but not enough for Chinese, Japanese, Korean, and certain others. The correspondence between characters and bit combinations is defined by the [Unicode](http://www.unicode.org) standard, which is beautifully described at www.unicode.org.

Representing Logical Values

The integer and floating-point formats represent numerical values. The `char` data type represents text characters. Java has one last data type, *boolean*, which represents logical values. Different JVMs may use different numbers of bytes to store booleans. Often 4 bytes are used, although this is not always the case.

The numerical and `char` types can represent many different values, from 256 possible values for byte all the way up to 18446744073709551616 for long. The `boolean` type can represent only two possible values: `true` and `false`. This data type is useful for controlling conditional execution. For example, a block of code might need to execute only if it's midnight and a certain database query returns more than 100 records but less than 500. Or a block of code might need to execute if the user has entered a special request and a password. Java uses logical values to express conditions like these that might be true and might be false. As you will see in [Chapter 3](#), there are special boolean operations that operate on these values.

Logical values and the operations that act upon them were first studied by George Boole, an 18th-century British mathematician. He is the only person in history whose name has been immortalized as a computer-language concept.

Recap of Java's Data Types

So far this chapter has introduced Java's 8 basic data types. These types are summarized in [Table 2.3](#).

Table 2.3: Java's Primitive Data Types

Data Type	# of Bits	Used For	Internal Format
byte	8	Very small integers	2's complement
short	16	Small integers	2's complement
int	32	Integers	2's complement
long	64	Large integers	2's complement
float	32	Fractions, very large numbers	Floating-point
double	64	Fractions, huge numbers	Floating-point
char	16	Characters	Unicode
boolean	??	Logic	Unavailable

These data types are collectively called *primitives* to distinguish them from object-oriented types. (We will begin our study of objects in [Chapter 7](#).)

We now turn to the question of what you can do with all this data.

Declaring and Assigning

In a sense, computer programming is the art of assigning the right value to the right data at the right time. In Java, as in many other languages, you have to declare your data before you use it. *Declaring* means telling the compiler the types of data you will be using. In this section you will see how to declare and assign data, and will look at your first complete Java program.

When you programmed SimCom, you had to specify the address of each data operand. That meant you had to remember what you were using the different memory bytes for. For example, the Times 5 program used byte #29 as a loop counter and byte #30 for storing the result. Yet, when you look at the program for the first time, it's very difficult to tell what's going on.

In Java, you never have to remember which memory location is being used for which purpose. In fact, there is no way to even *know* which memory location is being used for which purpose. You pick a name for each memory location you want to use, and you refer to memory locations by name rather by address. The compiler assigns the addresses. All you have to do is tell the compiler the names you will be using, and the data type associated with each name.

For example, if you wanted to use a byte as a loop counter, it would be reasonable to choose the name `loopCounter`. Then you would declare as follows:

```
byte loopCounter;
```

A piece of memory that is declared and named in this way is known as a *variable*, so we will use that term from here on.

A declaration has three parts: a data type, a name, and a semicolon.

The data type (for now) is one of the eight primitive types: byte, short, int, long, float, double, char, and boolean. Later we will introduce some other types.

The name has to begin with a letter, an underline (`_`), or a dollar sign (`$`). The rest of the name can consist of letters, underlines, dollar signs, or digits. It is good programming practice to use variable names that begin with lowercase letters. If the name consists of more than one word, the second word and all subsequent words begin with uppercase letters. This is what we have done with `loopCounter`. Later in this book, you will see that there are other entities besides variables for which you will assign names (including classes and interfaces). These entities use different naming conventions. Following the conventions helps make source code easy to read.

The semicolon is a vital part of a declaration. A declaration is a kind of *statement*. A statement is a single instruction. All statements must end with a semicolon. Otherwise, the compilation will fail and the compiler will print out an error message with the line number where it ran into trouble.

Be aware that it is inherently impossible to create a compiler that produces consistently helpful error messages. Imagine someone running along a rough cobblestone road. If his foot slips on a stone, he might stagger for a few steps before falling. Similarly, if the compiler slips on an ungrammatical line, it might stagger over a few more lines before crashing and printing a message.

For the sake of convenience, you can declare multiple variables in a single statement, as long as the variables are all of the same type. So the following:

```
double mass, velocity, energy;
```

is equivalent to the following:

```
double mass;
double velocity;
double energy;
```

After you declare a variable, you can assign values to it. The following two lines declare and assign a variable called `velocity`:

```
double velocity;
velocity = 123.456;
```

Notice that the assignment statement, like the declaration statement, ends with a semicolon. An assignment statement has the form `variable = value semicolon`. (In the [next chapter](#), you will see how the value can be a complicated mathematical formula. For now, the value will be a simple literal number.) Be aware that the equal sign is just a symbol, and its meaning is not exactly the same as its meaning in a mathematical context. In geometry, when we say $\text{Area} = \pi r^2$, the equal sign means "is, always has been, and always will be." In Java, the

equal sign means "store the value to the right of the equal sign in the variable to the left of the equal sign."

When you assign to a char variable, the easiest approach is to enclose the value in single quotes, like this:

```
char ch;
c = 'w';
```

After execution, the variable `ch` contains the Unicode representation for the letter `w`. The single quotes can also contain special codes, called *escape codes*, that encode special characters. The most useful of these are

- `'\n'` – Newline
- `'\t'` – Tab

A Very Simple Java Program

So far we have seen declaration and assignment lines, but only as code fragments. If you type any of the fragments into a file and try to compile the file, you will get nothing more than compiler error messages. This is because a well-formed Java program—even one that does almost nothing—must conform to certain structural rules.

And here we have a problem, which is best illustrated by an example. The following code listing is a complete Java program that contains a declaration and an assignment.

```
public class VerySimple
{
    public static void main(String[] args)
    {
        double age;
        age = 123.456;
    }
}
```

The problem is this: the program contains a number of words and symbols that have not yet been introduced, and that will require considerable explanation when the time comes. So for now, you just have to accept the mysterious parts of this code as things that must be done to make the program work.

Type the program into a file called `VerySimple.java`. Compile it by typing `javac VerySimple.java`. If you get compiler error messages, make sure you've typed in the program exactly as it appears here. The compiler output will be a file called `VerySimple.class`.

The preceding program appears just as it would in a source file. However, when listings of more than a few lines appear in print, it is convenient to number the lines:

```
1. public class VerySimple
2. {
3.     public static void main(String[] args)
4.     {
5.         double age;
6.         age = 12.34;
7.     }
8. }
```

The line numbers are convenient for referring to features of the code, but they should never appear in source code that is to be compiled. Here, the line numbers let us point out that the relevant parts of the listing are lines 5 and 6, and all the rest is mysterious code that will be explained later.

Output

The SimCom virtual machine lets you see all of memory all the time, but Java's memory is hidden. In the `VerySimple` program, there is no way to see the value of `age`. The following program declares and assigns `age`, and then prints its value to the console:

```
1. public class VerySimple2
2. {
3.     public static void main(String[] args)
4.     {
5.         double age;
6.         age = 12.34;
7.         System.out.println(age);
8.     }
9. }
```

The new line is #7. To print out any value, you can use the statement `System.out.println(theValue);`.

Here again, we ask you to accept that the syntax works. The explanation of *why* it works will come as soon as we have covered all the underlying concepts. For now, be aware that in order to print the value of any variable, you need to type that variable's name between the parentheses in a line like #7.

Notice that in line #1, the word following `class` has been changed from `VerySimple` to `VerySimple2`. The name following `class` has to match the name of the source file. Therefore, if you want to type in this program, you should store it in a file called `VerySimple2.java`. The compiler will generate an output file with the same name, followed by the `.class` suffix: `VerySimple2.class`. This compiler-output file is known as a *class file*. To run the application, type `java VerySimple2`. [Table 2.3](#) summarizes this naming consistency.

Table 2.3: Naming Consistency

Name in class line	Source filename	Class filename	Invocation
<code>VerySimple2</code>	<code>VerySimple2.java</code>	<code>VerySimple2.class</code>	<code>java VerySimple2</code>

Printing out the value of a variable is convenient, but it would be even more convenient to print out a reminder of what the value represents. "Age is 12.34" is much more informative than "12.34." The following program prints out the more informative line:


```
1. public class VerySimple3
2. {
3.     public static void main(String[] args)
4.     {
5.         double age;
6.         age = 123.456;
7.         System.out.println("Age is " + age);
8.     }
9. }
```

In line #7, the text inside the double quotes is known as a *literal string*. The plus sign does not indicate addition, since adding text to a number doesn't really mean anything. In this context, the plus sign just means that the literal string is to be printed out, followed by the value of `age`. Within the parentheses of a `println` statement, you can have any number of alternating literal strings and variables. So if you wanted to print out the values of variables `i`, `j`, and `k`, separated by commas, you could use the following line:

```
System.out.println(i + "," + j + "," + k);
```

Now that you can declare, assign, and display variables, you are ready for the next step: mathematical operations. That is the topic of the [next chapter](#).

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. According to [Table 2.1](#), the maximum values for the byte and short data types are 127 and 32767, respectively. Use the Twos-Complement Lab animated illustration to verify this. Which byte and short bit patterns produce the maximum values? In general, which bit pattern produces the maximum value for a two's complement number of N bits?
2. According to [Table 2.1](#), the minimum values for the byte and short data types are -128 and -32768, respectively. Use the Twos-Complement Lab animated illustration to verify this. What byte and short bit patterns produce the minimum values? In general, what bit pattern produces the minimum value for a two's complement number of N bits?
3. Launch the Twos-Complement Lab animated illustration by typing `java TwosCompLab`, set the data type to int, and set all the bits to 1. Then set the three bits on the right to 0. Compute the value. Do the same for the byte and short data types. What do you observe?
4. Launch the Floating-Point Lab animated illustration by typing `java floating.FloatFrame`. Set the rightmost bit to 1 and all other bits to 0. The value represented is 1.4E-45. Try changing various bits' values by clicking on them. Can you create a value that is smaller than 1.4E-45 but still greater than 0?
5. Write a Java application that declares and assigns values to three int variables named `x`, `y`, and `z`. Print out all three values, separated by commas, on a single line.
6. *White space* means spaces, tabs, and line-break characters. Type in the VerySimple application from this chapter (reproduced below) and experiment with inserting white space. Does anything change during compilation or execution if you insert extra spaces between `public` and `class`? What if you insert a line break between `public` and `class`? Can you find any adjacent words or symbols such that inserting white space between them changes compilation or execution?

```
public class VerySimple
{
    public static void main(String[] args)
    {
        double age;
        age = 123.456;
    }
}
```

Chapter 3: Operations

The [previous chapter](#) showed you how to declare various types of primitive variables, and how to assign and print out the values of variables. This chapter will look at the computational operations that you can perform on Java data. For numeric data—that is, for all primitive types other than `boolean`—these computations include the familiar arithmetic operations of addition, subtraction, multiplication, and division, as well as some more exotic operations. Arithmetic operations are not applicable to `boolean` data, which has its own group of operations.

Before covering these topics, we will look at two ways to make programs easier to read: white space and comments.

White Space and Comments

This chapter is going to present some techniques for writing long and intricate programs. Before we begin, though, let's look at how to write programs that are easy to read and understand.

Consider the following lines of code:

```
int x;  
x = 5;
```

As far as the Java compiler is concerned, this code is identical to the following:

```
int x;      x=5;
```

These two versions produce exactly the same compiled bytecode. In the preceding line, the gap after the semicolon can be created by adding a few tabs or a lot of spaces. Either way, it doesn't matter to the compiler.

The compiler ignores any blank space created by using the spacebar or by typing Tab or Enter. Such space is called *white space*. You can use white space to make your source code more readable. For example, the following code declares four variables:

```
double velocity; boolean b;  
short x;  
  
long  
  
hotSummer;
```

If you use this code in a program, someone who is unfamiliar with the program will have a hard time figuring out what your intention is. (And the person sweating over your source code could be yourself, reviewing your own code long after you originally wrote it.)

You can make your code more readable to humans by manipulating white space, like this:

```
double velocity;  
boolean b;  
short x;  
long hotSummer;
```

Or better yet:

```
double    velocity;  
boolean   b;  
short     x;  
long      hotSummer;
```

In the first example, some spaces and a return have been removed, and a return has been added, so that all the left edges line up. In the second example, white space has been added. Now the eye of any reader, including you, will subconsciously arrange the code into two columns. The words on the left are all data types, and the words on the right are all variable names. The columns are easily aligned: You just press Tab before each data type and before each variable name. This formatting scheme is so clear that it is considered correct style by convention. Any other formatting arrangement would be less readable, and would also be considered sloppy style.

The [previous chapter](#) presented the following simple program:

```
1. public class VerySimple  
2. {  
3.     public static void main(String[] args)  
4.     {  
5.         double age;  
6.         age = 12.34;  
7.     }  
8. }
```

Note Remember that the line numbers are not part of the source code. They are just included to make it easier to refer to particular lines.

A Java program consists of one or more class definitions (though we will not discuss class definitions until [Chapter 7, "Introduction to Objects"](#)). For now, be aware that lines 2-8 are the definition of a class named `VerySimple`. The class definition begins with an open curly bracket (line 2) and ends with a closed curly bracket (line 8). Since these brackets are vertically aligned in the same column, our brains notice that they are spatially related, and we subconsciously assume that they must also be functionally related.

A class definition can contain (among other things) a number of method definitions. Methods will be discussed in detail in [Chapter 4, "Methods"](#); for now, you just need to be aware that lines 4-7 are the definition of something called a method, whose name is `main`. The method definition begins with an open curly bracket (line 4) and ends with a closed curly bracket (line 7). Again, the vertical alignment of the brackets gives us visual information about the structure of the program.

Notice how easy it is to look at lines 3 and 4, which tell us, "the method starts here," and find the end of the method. Within the method, all the code (lines 5 and 6) is vertically aligned. If you look at the listing with your eyes out of focus, all you see are several levels of nested blocks of blurry stuff. A Java program is (mostly) a block that contains blocks that contain blocks, etc. It is extremely important to use white space and indentation to indicate the nesting level of all your lines of code.

In addition to white space, the Java compiler also ignores *comments*. There are two kinds of comments: single-line and multi-line.

A single-line comment begins with two slashes (`//`). There can't be anything between the slashes. The compiler ignores everything from the slashes through the end of the line. This lets you put descriptive text after the slashes. Usually, the text explains what just happened in the line. For example:

```
float distance; // Units are microns
double weight; // Units are ounces
```

Note the use of white space to vertically align the comments.

A multi-line comment, also known as a *traditional* comment, can span more than one line but doesn't have to. This kind of comment begins with a slash immediately followed by an asterisk (`/*`). The comment ends with an asterisk immediately followed by a slash (`*/`). For example:

```
/* Declare and initialize variables that
will later be used for computing
time-distortion effects at relativistic
speeds. All distance units are miles,
not kilometers. */
```

```
double speedOfLight;
int numberOfPlanets;
```

```
speedOfLight = 186000;
numberOfPlanets = 9;
```

Note the use of blank lines to separate the multi-line comment from the declarations, and the declarations from the assignments.

Now that you know how to make source code easy to read, we can move on to the main topic of this chapter, which is how to write a program that makes your computer actually compute something.

Arithmetic Operations

Java's arithmetic operations fall into two categories: basic arithmetic (addition, subtraction, multiplication, and division), and some more exotic operations such as modulo and shifting. We will begin by looking at the simple operations.

Basic Arithmetic

The following code computes and prints out the sum, difference, product, and quotient of two numbers:

```
public class C3
{
    public static void main(String[] args)
    {
        int    x, y;           // Inputs
        int    sum;           // x plus y
        int    diff;          // x minus y
        int    product;       // x times y
        int    quotient;      // x divided by y

        /* First assign initial values to
           the x and y inputs. */
        x = 12;
        y = 3;

        // Now do arithmetic.
        sum = x + y;
        System.out.println("sum = " + sum);
        diff = x - y;
        System.out.println("diff = " + diff);
        product = x * y;
        System.out.println("product = " + product);
        quotient = x / y;
        System.out.println("quotient = " + quotient);
    }
}
```

Note the use of the asterisk (*) to indicate multiplication. The other three symbols (+, -, and /) are recognizable from standard arithmetic. These symbols (as well as a few others that we'll see later on in this chapter) are known as *binary operators*. Here the word *binary* indicates that the operators work on two numbers at a time, known as *operands*. Most Java operators are binary, but there are several *unary operators* that each take a single operand. There is even a *ternary operator* that takes three operands. (We will postpone discussion of the ternary operator until [Chapter 5, "Conditionals and Loops."](#))

There is nothing surprising about this program's output:

```
sum = 15
diff = 9
product = 36
quotient = 4
```

As mentioned in the [previous chapter](#), the meaning of the equal sign (=) here is a bit different from its traditional mathematical meaning. In Java, the equal sign is called the *assignment operator*. It tells the computer to compute the value on the right-hand side of the equal sign (usually abbreviated *rhs*), and to store the result in the variable that appears on the left-hand side (usually abbreviated *lhs*). Until now, the *rhs* has been a literal value, but as this program shows, the *rhs* can also be a calculation. The calculation's arguments can be variables or literals, so the following lines would be valid:

```
int halfProduct;
halfProduct = product / 2;
```

Java allows you to declare a variable and assign its initial value, all in a single statement. The preceding code can be rewritten as

```
int halfProduct = product / 2;
```

Later you can assign a different value to `halfProduct`. You can reassign values to variables as often as you like. Just don't declare the variable more than once, because the second declaration will cause a compiler error.

The *lhs* of an assignment can appear in its own *rhs*. Consider the following line:

```
x = x + 5;
```

If this were a line of algebra and not a computer-language statement, it would be ridiculous. When you subtract *x* from both sides, you get $0 = 5$. But in Java, it is perfectly legal because the equal sign means assignment. The line says to add the value of *x* plus 5 and store the result back in *x*.

Precedence and Parentheses

Multiple operations can be combined in a single statement. For example, you might use the following code to compute the area of a circle whose radius is known:

```
double area = 3.14159 * r * r; // Pi-r-squared
```

Use caution when combining different operators in a single statement. It would be reasonable to expect the statement to be evaluated left to right, but Java doesn't do it that way. For example, you might expect that after the following line executes, the value of *x* is 502:

```
int x = 1000 + 4 / 2;
```

Actually, *x* is 1002. Java gives multiplication and division higher *precedence* than addition and subtraction. This means that in a

statement such as the one above, any multiplication or

division is performed before any addition or subtraction, even if the addition and subtraction appear first. So the division happens first ($4/2 = 2$), and then the addition ($1000+2 = 1002$).

Note Java has strict evaluation precedence rules that govern all the operations presented in this chapter. The precedence is summarized in [Table 3.6](#).

If you don't like a statement's order of evaluation, as dictated by the precedence of its operators, you can use parentheses. Operations that appear in parentheses have higher precedence than operations that do not. So the following code really does compute a result of 502:

```
int x = (1000 + 4) / 2;
```

Parentheses can be nested, as the following example shows:

```
int x = 1+(2*(3-4)+(5-6)*(7+8));
```

The result is -16.

The EvaluatorLab animated illustration will help you get used to parentheses and operator precedence. To launch the program, type `java eval.EvaluatorLab`. You will see the display shown in [Figure 3.1](#).



Figure 3.1: EvaluatorLab

Type any arithmetic expression into the text field and press Enter. The arithmetic expression can consist of any combination of literal integers, parentheses, and the binary operations `+`, `-`, `*`, and `/`. Click on the Run button to see an animation of the evaluation of the expression. Click on Step to see an animation of just the next step in the expression's evaluation. The Run Lightspeed and Step Lightspeed buttons perform the evaluation immediately, without animation. [Figure 3.2](#) shows the program after evaluating the configuration of [Figure 3.2](#).



Figure 3.2: EvaluatorLab after evaluation

Type the following expressions into EvaluatorLab and observe the results:

- $1000+4/2$
- $(1000+4)/2$
- $1+(2*(3-4)+(5-6)*(7+8))$

The EvaluatorLab only works with integer data. In Java, integer addition, subtraction, and multiplication behave exactly as you would expect. Division, however, has a problem. Dividing an integer by an integer can produce a non-integer result. Be aware that when Java divides a byte, short, char, int, or long by a byte, short, char, int, or long, the result is *truncated*. This means that any fractional part is discarded. For example, $48 / 10$ would be truncated from 4.8 to 4.

Truncation may seem like a problem, but it really isn't. If you are going to be dividing, and you know that the fractional parts of the results will be important, just use a floating-point data type (float or double) rather than an integer type. A good rule of thumb is to use integer types for quantities that can be counted, such as the number of employees or grizzly bears, and to use floating-point types for things that can be measured, such as weight or speed.

Bitwise Operations

A *bitwise* operation treats its operands as collections of individual unrelated bits, rather than as representations of numbers. You can only perform bitwise operations on integer data. Floats and doubles are not allowed.

There is one unary bitwise operator. Its symbol is the tilde (`~`). It toggles all the bits of its operand, changing all 0s to 1s and all 1s to 0s. [Figure 3.3](#) illustrates the operation `~144`.

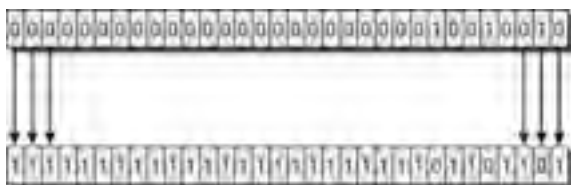


Figure 3.3: The unary bitwise operator ~

With a binary bitwise operation, the *n*th bit of the result is computed from the *n*th bits of the two operands. The three binary bitwise operations are and, or, and exclusive or. The operator symbols are &, |, and ^.

The "and" of two bits is 1 if both bits are 1. Otherwise, the result is 0. Another way to say this is that the result is 1 if one argument bit is 1 *and* the other argument bit is 1.

The "or" of two bits is 1 if either (or both) of the bits is 1. Otherwise, the result is 0. Another way to say this is that the result is 1 if one argument bit is 1 *or* the other argument bit is 1 (or both).

The "exclusive or" of two bits is 1 if either (but *not* both) of the bits is 1. Otherwise, the result is 0.

Table 3.1 shows the results of the three binary bitwise operations on all possible combinations of operand bits a and b.

Table 3.1: Binary Bitwise Operations

	a&b	a b	a^b
a,b = 0,0	0	0	0
a,b = 0,1	0	1	1
a,b = 1,0	0	1	1
a,b = 1,1	1	1	0

As you can see from the table, the only way for & to generate a 1 is if both operands are 1. The only way for | to generate a 0 is if both operands are 0. ^ generates a 1 if its two operands are different.

In practice, the binary bitwise operators work on integer values, not on integer bits. For example, if you take the "and" of two ints, bit 0 of the result will be the "and" of the bit 0s of the two operands. Bit 1 of the result will be the "and" of the bit 1s of the two operands, and so on through bit 31. This is illustrated in Figure 3.4.

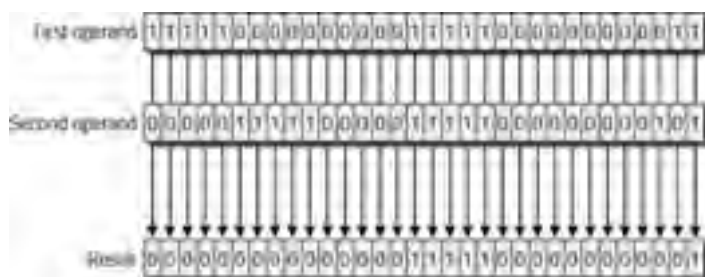


Figure 3.4: Bitwise "and"

As you can see, every bit in the result is computed solely from the corresponding bits in the two operands.

Modulo

Java supports some binary arithmetic operations that we don't often encounter outside the realm of computer programming: the modulo operation and three shift operations.

The symbol for modulo is the percent sign (%). The operation divides the first operand by the second operand and returns the remainder. So for example, 506 % 100 is 6, 507 % 100 is 7, and so on.

Shifting

Shifting operations move the bits of an integer operand to the left or right by some number of positions. There is one left-shift operation; its symbol is <<. There are two right-shift operations; their symbols are >> and >>>.

The left-shift operation is straightforward. The first operand is the value to be shifted. The second operand is the number of bit positions to shift by. Figure 3.5 illustrates 18 << 5.

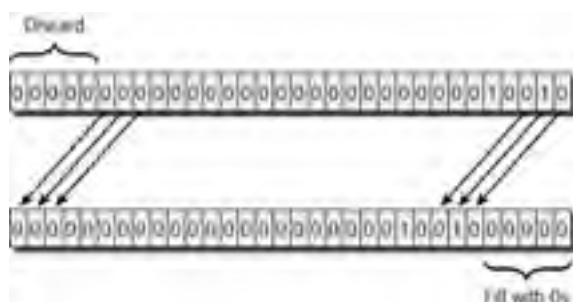


Figure 3.5: Left-shift: <<

As the figure shows, the high-order bits of the shifting value are discarded. The low-order bits are all set to 0.

The result of the shift in base-2 is 1001000000, which is 576. Is there any numerical relationship between 576 and the original value of 18? Yes, 576 is 18 times 32. Is there anything special about 32? Yes, 32 is 2 raised to the power of 5. In general, left-shifting x by y is the same as multiplying x by 2^y . This is elegant, and it makes good sense. Left-shifting a value by, for example, 3 bit positions is like writing three 0s to the right of the number. In base-10, if you write three 0s to the right of a number, you have multiplied that number by 10^3 (that is, by 1000). It is not surprising that something similar happens in base-2.

There are two right-shift operations. One of them is bitwise, and the other is numeric.

The bitwise right-shift (\gg) is just the opposite of the left-shift: bits are moved to the right, any bits that fall off the right end are lost, and the left end is filled with 0s. This is illustrated in [Figure 3.6](#).

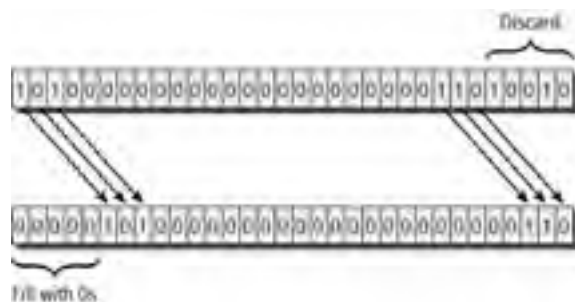


Figure 3.6: Bitwise right-shift: \gg

The original value in the figure has its sign bit set to 1, representing a negative number. The result has a sign bit of 0, since the \gg operation always shifts 0s into the left portion of the result. You can see that \gg always converts negative numbers to positive numbers that have no clear relationship to the original values. This is why the \gg shift is called *bitwise*. All it does is move bits.

The other shift operation is \gg . It is different from \gg in only one respect: The left bits of the result are set to the sign bit of the original value, instead of being always set to 0. For positive numbers, the original sign bit is 0, so \gg is the same as \gg . But for negative numbers, the result is very different, as [Figure 3.7](#) shows.

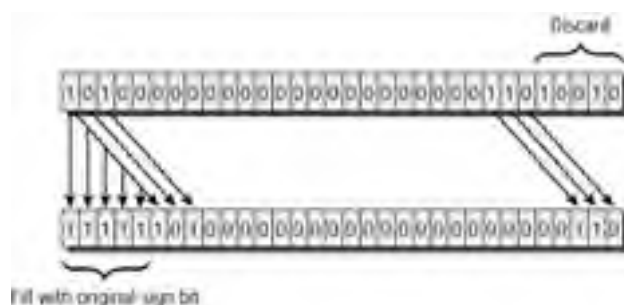


Figure 3.7: Numeric right-shift: \gg

The sign of the result is always the sign of the original value. Does the result have any numerical relationship to the original? Yes, although it is hard to see the relationship when you look at [Figure 3.7](#). It turns out that $x \gg y$ is the same as $x / 2^y$.

The different right-shift operations can be confusing until you have some experience with them. The ShiftLab animated illustration will help you get that experience. Launch the program by typing `java shift.ShiftLab`. The display shows a 32-bit int value to be shifted, as illustrated in [Figure 3.8](#).

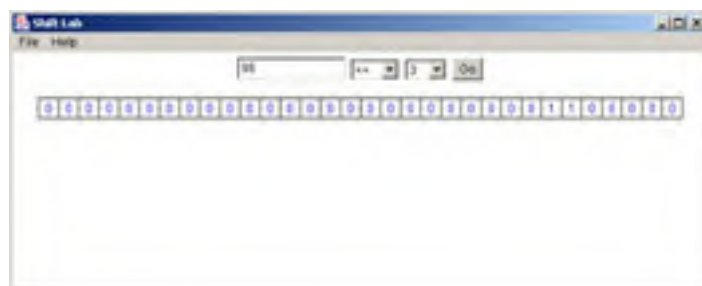


Figure 3.8: ShiftLab

You can change the value by typing a base-10 number (positive or negative) into the text field, or by clicking on individual bits in the display. Select the desired shift operation (\ll , \gg , or \gg) and the desired shift size, and then click on the Go button. The program will animate the shift that you've specified. [Figure 3.9](#) shows the result of "96 \ll 3".



Figure 3.9: ShiftLab after shifting

Try viewing the following shifts:

```
10 << 10
16384 >> 14
-1 >>> 1
-1 >> 1
-1 >> 20
-2147483648 >>> 31
```

Unary Arithmetic

The unary arithmetic operators have the symbols + and -. These are the same as the symbols for binary addition and subtraction, so the compiler has to figure out from context which kind of operation you want. A + or - between two operands is a binary operator; a + or - with no operand to the left is unary.

The unary - operation just changes the sign of its operand. So for example, the following code prints out $y = -5$:

```
int x = 5;
int y = -x;
System.out.println("y = " + y);
```

The unary + operator maintains the sign of its operand. In other words, it doesn't really do anything.

++ and --

Two of the most common operations in programming are adding or subtracting 1 with a variable, and storing the result back in the variable. If the variable is called x , these operations can be programmed as follows:

```
x = x + 1;
x = x - 1;
```

However, Java provides some convenient abbreviations. The first line can be abbreviated in either of the following ways:

```
x++;
++x;_
```

The second line can be abbreviated in either of the following ways:

```
x--;
--x;
```

The following program shows these operators in action:

```
public class PlusPlusMinusMinus
{
    public static void main(String[] args)
    {
        int x = 10;
        int y = x;
        x++;
        y--;
        System.out.println("x=" + x + ", y=" + y);
    }
}
```

The output is

```
X=11, y=9
```

You can see that x has been incremented and y has been decremented.

When the operator appears after the operand, the rhs is first calculated as if the operator were not present. Then the rhs value is assigned to the lhs. Lastly, the operand of ++ or -- is incremented or decremented. For example:

```
public class PostDec
{
    public static void main(String[] args)
    {
        int x = 10;
        int y = 1000 + x--;
        System.out.println("x=" + x + ", y=" + y);
    }
}
```

The output is

$x=9$, $y=1010$

You can see that x , which was originally 10, must have been added to y before being decremented.

When $++$ appears before its argument, it is called the *pre-increment* operator. When it appears after its argument, it is called the *post-increment* operator. Similarly, $--$ before its argument is called the *pre-decrement* operator, and $--$ after its argument is called the *post-decrement* operator.

Team LIB

4 PREVIOUS

NEXT 5

Boolean Operations

So far, all the operations we have looked at have dealt with numbers. Now we turn our attention to operations that work on boolean data. Some of these operations share symbols with similar numeric operations (`|`, for example). However, the boolean versions are essentially different from their numeric counterparts.

Most of Java's boolean operations are binary, and both operands must be of boolean type.

And, Or, Exclusive Or, Inversion

We have already seen these as bitwise arithmetic operations. The symbols for and, or, and exclusive or are, as before, `&`, `|`, and `^`, respectively. The symbol for inversion is `!` rather than `~`.

The following program prints out the results of applying these operators to `true` values:

```
public class BooleanOps
{
    public static void main(String[] args)
    {
        boolean a = true;
        boolean b = true;
        boolean x = a & b;
        System.out.println("true&true = " + x);
        x = a | b;
        System.out.println("true|true = " + x);
        x = a ^ b;
        System.out.println("true^true = " + x);
        x = !a;
        System.out.println("!true = " + x);
    }
}
```

The output is

```
true&true = true
true|true = true
true^true = false
!true = false
```

Boolean operations, like arithmetic operations, have precedence. The unary `!` operator is evaluated before the binary `&`, `|`, and `^`. For example, the value of `!false|true` is `true`, because `!false` is evaluated first. You can override the effects of precedence by using parentheses. In the current example, if you want the `|` operator to execute before `!`, use the expression `!(false|true)`.

The BoolLab animated illustration demonstrates the evaluation of boolean expressions. Launch the program by typing `java bool.BoolLab`. [Figure 3.10](#) shows the program just after it starts up.

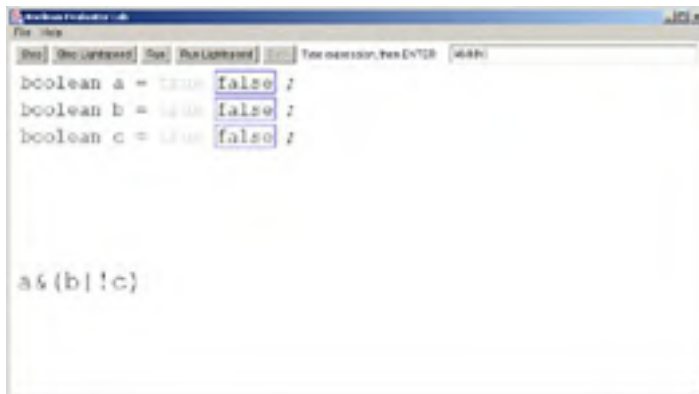


Figure 3.10: BoolLab: initial screen

You can type into the text field any valid expression composed of the variables `a`, `b`, and `c`, the literals `true` and `false`, the operators `&`, `|`, `^`, and `!`, and parentheses. After you enter the expression you want, press Enter. The expression will appear in large font in the main area of the window. As with EvaluatorLab, you can click on the Run button to see an animation of the expression being evaluated. Click on Step to see an animation of just the next step in the expression's evaluation. The Run Lightspeed and Step Lightspeed buttons perform the evaluation immediately, without animation. [Figure 3.11](#) shows the result of running the configuration of [Figure 3.10](#).



Figure 3.11: BoolLab after execution

Try the following expressions in BoolLab:

```
false | false | false | false | false | true
true & true & false & true & true
false & (((!(true ^true) & (false|true))|false)^false)
true | (((!(false ^ false) & (true | false))| true)^ true)
```

The first expression shows that when you take the or of a number of values, a single true is enough to make the entire result true. The second expression shows that when you take the and of a number of values, a single false is enough to make the entire result false.

Now that you have seen Java's simple boolean operators, let's move on to the short-circuit operators, which shorten the time it takes to execute an operation. Before you read the [next section](#), can you guess the point of the lengthy third and fourth expressions in the preceding code?

Short-Circuit Operators

This really happened to me, and perhaps it has happened to you. When I was a little boy, I was allowed to go out and play if I had made my bed and finished my homework. I didn't mind doing my homework, but I hated making my bed and often I wouldn't do it. When my mother asked if I had made my bed, I would start to say, "No, but I ..." I was going to say that I had done my homework, but my mother would interrupt me. She was a busy person and she had heard all she needed to hear. Our agreement was that I would do two chores. As soon as she knew that I had not done *one* of those chores, there was nothing I could say about the *other* chore to convince her that I had lived up to my part of the agreement.

False & anything is false. When you compute $x \& y$, and x is false, you don't have to spend any time at all on y . You already know the answer.

Consider the following expression, which you were invited to type into BoolLab in the [previous section](#):

```
false & (((!(true ^true) & (false|true))|false)^false)
```

At first glance, this expression looks so complicated that you would not want to figure out its value in your head. But at second glance, once you realize that the expression's form is "false & anything," you don't have to look any further. The value is false, no matter what comes after the &.

Java provides an alternative to the & operator. It is called the *short-circuit &* operator, and its symbol is &&. The short-circuit version stops computing and immediately returns false if its first operand is false. Let's slightly modify the previous example:

```
false && (((!(true ^true) & (false|true))|false)^false)
```

Now the first operator is the short-circuit version. This expression evaluates to the same value as the previous version, but the evaluation takes less time because everything between the outermost parentheses is ignored.

There is also a short-circuit version of the | operator. Its symbol is ||, and it immediately returns true if its first operand is true.

The BoolLab animated illustration supports short-circuit operations. Launch the program again (type `java bool.BoolLab`), and see how it evaluates the following expressions:

```
false && (true|false)
true && (true|false)
false || && (true|false)
true || && (true|false)
```

Java's short-circuit operators allow you to profit from the principle that false-and-anything is false and true-or-anything is true. The amount of profit may seem trivial. In this example, the processing time that's saved by using && could not possibly be more than a microsecond or so. But a short-circuit expression might be executed not once but many times—even many millions or billions of times—so any time savings will be significant.

You will learn how to execute a single expression multiple times when you look at loops in [Chapter 5](#). Moreover, the second operand of the short-circuit operator might be a call to a method that takes minutes or hours to execute. In this case, you definitely do *not* want to process the second operand unless you really have to. The [next chapter](#) will look at methods and method calling.

Now let's look at Java's comparison operators. These are binary operators whose operands can be numeric or boolean. The result type is always boolean.

Comparison Operations

Java's comparison operators always return a boolean value. Most of these operators work on numeric operands, but there are two that can take numeric or boolean operands. [Table 3.2](#) summarizes the comparison operators.

Table 3.2: Comparison Operators

Operator	Meaning	Numeric Operands	Boolean Operands
==	Equals	3	3
!=	Does not equal	3	3
>	Is greater than	3	no
>=	Is greater than or equal to	3	no
<	Is less than	3	no
<=	Is less than or equal to	3	no

Note that the symbol for the equals comparison operator is a double equal sign (==), to distinguish it from the assignment symbol (=).

Comparison operators can be combined with other boolean operators. For example, assuming *w*, *x*, *y*, and *z* are variables of some numeric type, you might use the following expression:

```
w == x | y < z
```

Comparison operators have higher precedence than boolean operators, so the == and < comparisons happen before the | is evaluated. The example can be rewritten as follows:

```
(w == x) | (y < z)
```

The parentheses make the expression clearer without changing the order of computation. Expressions such as this one are most often seen in flow-control statements, which allow you to execute blocks of code repeatedly, or only if certain desired conditions are met. We will look at flow-control statements in [Chapter 5](#).

Compound Assignment

A very common practice is to perform an operation on the value of a variable and then store the result back in the variable. For example, you might want to do the following:

```
x = x - y;
```

For situations like this, Java provides an abbreviation called *compound assignment*. A compound assignment lets you modify a variable (in certain restricted ways) and store the value back in the variable, all in a single statement. Compound assignments have the form *variable op= expression*, where *op* is a binary operation symbol that is immediately followed by `=`.

Table 3.3 summarizes the compound assignment operators. (The table assumes that *b* is boolean and *x* is of some numeric type.)

Table 3.3: Compound Assignment

Operator	Example	Equivalent
<code>+=</code>	<code>x += 5;</code>	<code>x = x+5;</code>
<code>-=</code>	<code>x -= 5;</code>	<code>x = x-5;</code>
<code>*=</code>	<code>x *= 5;</code>	<code>x = x*5;</code>
<code>/=</code>	<code>x /= 5;</code>	<code>x = x/5;</code>
<code>%=</code>	<code>x %= 5;</code>	<code>x = x%5;</code>
<code><<=</code>	<code>x <<= 5;</code>	<code>x = x<<5;</code>
<code>>>=</code>	<code>x >>= 5;</code>	<code>x = x>>5;</code>
<code>>>>=</code>	<code>x >>>=5;</code>	<code>x = x>>>5;</code>
<code>&=</code>	<code>b &= false;</code>	<code>b = b&false;</code>
<code> =</code>	<code>b = false;</code>	<code>b = b false</code>
<code>^=</code>	<code>b ^= false;</code>	<code>b = b^false;</code>

Compound assignments provide no new functionality. They just provide a convenient way to abbreviate.

Numeric Result Type

You have now learned about all of Java's unary and binary operators. Before closing this chapter, you need to learn about more topic: the result type of numeric operations.

Clearly, it would be inconvenient to prohibit operating on mixed types. You may find yourself doing arithmetic on two numbers, one of which might be an int and the other of which might be a float. The next issue to consider is the data type of the result.

Java's rules for determining the type of the result are based on the concept of *width*. As you saw in [Chapter 2](#), every numeric type has a range. This is shown in [Table 3.4](#).

Table 3.4: Ranges of Numeric Types

Name	Size	Minimum Value	Maximum Value
byte	8 bits	-128	127
short	16 bits	-32768	32767
char	16 bits	0	65535
int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807
float	32 bits	-3.4×10^{38}	3.4×10^{38}
double	64 bits	-1.8×10^{308}	-1.8×10^{308}

If the range of any type completely contains the range of another type, the first range is considered to be *wider* than the second range. If a range is completely contained within another range, the first range is *narrower* than the second range. [Table 3.4](#) shows that the byte type is narrower than the short type. Note that some types are neither wider nor narrower than some other types. Short, for example, is neither wider nor narrower than char.

[Figure 3.12](#) illustrates data type width. [Figure 3.12](#) is definitely *not* drawn to scale. If the line representing double were scaled to the line representing byte, the double line would be 5×10^{275} light years long. I really wanted to print the line to scale, because I believe accuracy is important, but my editor pointed out that the line would be 3×10^{273} times the diameter of the universe. The publisher was unwilling to pay for that much ink, and economics won out.

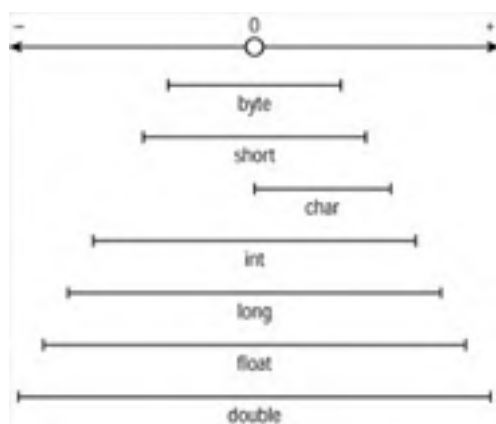


Figure 3.12: Data type width, not to scale

Another way to imagine width is shown in [Figure 3.13](#). A type is wider than another type if you can get from the first type to the second type by following the arrows. So double is wider than byte, and long is wider than char.

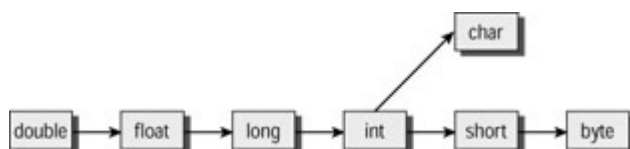


Figure 3.13: Data type width relationships

[Figure 3.13](#) shows that float is wider than long, even though longs are 64 bits and floats are only 32 bits. That might seem backwards, but you'll see why it's true if you think about the definition of "*wider*." If you don't feel like thinking about that right now, you can wait until you get to Exercise 6.

Java's rule for the result data type is this: It's either int or the type of the widest operand, whichever is wider. This means that the result of any arithmetic operation will never be a byte, short, or char.

This rule applies to unary as well binary operations. For example, if s is a short, -s is an int.

Table 3.5 summarizes the result type combinations for binary operations.

Table 3.5: Binary Arithmetic Result Types

	byte	short	char	int	long	float	double
byte	int	int	int	int	long	float	double
short	int	int	int	int	long	float	double
char	int	int	int	int	long	float	double
int	int	int	int	int	long	float	double
long	long	long	long	long	long	float	double
float	float	float	float	float	float	float	double
double	double	double	double	double	double	double	double

It is important to know about arithmetic result types because of another rule: You can only assign a numeric value to a variable whose type is the same as, or wider than, the type of the numeric value. If you try to do anything else, the compiler will generate an error. This makes sense, because you might be trying to store a value that the variable cannot represent. For example, you can't store a long value in a byte variable, because the long value might be greater than 127 or less than -128. So the following code fragment will generate a compiler error:

```
long distance = 999999;  
long time = 5000;  
byte rate = distance / time;
```

This rule sometimes gets in your way when you just want to initialize a variable with a literal value. Java dictates that all floating-point literals are doubles, and all integral literals are ints. So 3.14 and 2.5e33 are both doubles, and 1234 is an int.

If you try to assign a value like 3.14 to a float (such as `float f = 3.14;`), the compiler will complain that you are trying to assign a double to a float. To fix the problem, append the letter f or F to the end of the literal number. This will tell the compiler that the literal is really a float:

```
float f = 3.14f; // Or 3.14F
```

The situation is a bit stranger if you try to assign a big literal value to a long variable. The following line generates a compiler error:

```
long timeAgo = 999999999999; // 12 digits
```

The 12-digit string of 9s is too big to be represented by an int. Even though you innocently want to assign a big number to a long variable, behind the scenes the compiler is going to try to create an int to store the value 999999999999. This is because the compiler uses ints to store literal integral numbers. To get around the problem, append the letter l or L to the literal value to indicate that it's really a long:

```
long timeAgo = 999999999999L;
```

You could also use 999999999999l, but a lowercase l looks too much like a 1. The uppercase version is definitely preferable.

What happens when you want to assign a literal value to a byte, short, or char variable? In this case, the compiler gives you a break. As long as the literal value falls within the variable's range, statements such as these are legal:

```
byte x = 12;  
short y = -22;  
char z = 0;
```


Precedence Summary

This chapter presented 16 Java operations. Their evaluation precedence is shown in [Table.3.6](#). Higher-precedence operators, the ones that are evaluated first, appear at the top.

Table 3.6: Operator Precedence

Category	Operators
Unary	+ - ! ~ ++ --
Higher-precedence arithmetic	* / %
Lower-precedence arithmetic	+ -
Shift	>> << >>>
Bitwise	& ^
Short circuit	&&

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. What happens when a comment appears inside a literal string? (Recall from [Chapter 2](#) that a literal string is a run of text enclosed between double quotes.) What would the following line of code do?

```
System.out.println("A /* Did this print? */ Z");
```

Write a program that includes this line. Does the program print the entire literal string, or does it just print "A Z"?

2. What is the value of ~ 100 ? What is the value of $\sim \sim 100$? First try to figure it out, and then write a program to print out the values. (Hint: You can figure it out without using pen and paper if you remember something that was discussed in [Chapter 2](#).)

3. Write a program that prints out the following values:

```
32 << 3
```

```
32 >> 3
```

```
32 >>> 3
```

```
-32 << 3
```

```
-32 >> 3
```

```
-32 >>> 3
```

4. What are the values of the following expressions? First do the computations mentally. Then write a program to verify your answer.

```
false & ((true^(true&(false|!(true|false))))^true)  
true | (true^false^false^true&(false|!(true&true)))
```

5. The following expression looks innocent:

```
boolean b = (x == 0) | (10/x > 3);
```

You can assume x is an int. Write a program that prints out the value of this expression for the following values of x : 5, 2, 0. What goes wrong? (You will see a failure message that you might not be familiar with, because we have not introduced it yet. Don't worry – just try to understand the general concept.) How can you make the code more robust by adding a single character to the expression?

6. The 32-bit float type is wider than the 64-bit long type. How can a 32-bit type be wider than a 64-bit type?
7. Write a program that contains the following two lines:

```
byte b = 6;  
byte b1 = -b;
```

What happens when you try to compile the program?

Chapter 4: **Methods**

Overview

So far, all of the Java applications we have seen have been linear: The processing has proceeded line by line, from the start through the end of the block that begins `public static void main(String[] args)`.

Such applications don't really take full advantage of your computer's capability. In fact, with linear code your computer is not much more than an expensive calculator. The topics in the following two chapters will begin to branch out—and so will the paths of execution through the programs we will study. Instead of proceeding line by line, we will see how to make the execution path detour, fork, and loop.

This chapter will look at methods, which are detours in the path of execution. [Chapter 5, "Conditionals and Loops,"](#) will introduce statements that redirect the flow of the program.

Method Structure

The easiest way to introduce methods is with an example. Suppose you want to print out the fifth powers of the numbers 5 through 9. The following code, which doesn't use methods, does the job in a clumsy, inelegant way:

```
1. public class NoMethods
2. {
3.     public static void main(String[] args)
4.     {
5.         int n = 5;
6.         int n5th = n*n*n*n*n;
7.         System.out.println(n + " >=> " + n5th);
8.         n = 6;
9.         n5th = n*n*n*n*n;
10.        System.out.println(n + " >=> " + n5th);
11.        n = 7;
12.        n5th = n*n*n*n*n;
13.        System.out.println(n + " >=> " + n5th);
14.        n = 8;
15.        n5th = n*n*n*n*n;
16.        System.out.println(n + " >=> " + n5th);
17.        n = 9;
18.        n5th = n*n*n*n*n;
19.        System.out.println(n + " >=> " + n5th);
20.    }
21. }
```

The application's output is

```
5 >=> 3125
6 >=> 7776
7 >=> 16807
8 >=> 32768
9 >=> 59049
```

When you look at the source code, you might get the feeling that life ought to be better than this. The application has a lot of repetition... and aren't computers supposed to be good at eliminating repetitive tasks? Five of the lines (6, 9, 12, 15, and 18) do almost the same computation, but not quite. Each of them multiplies something by itself 5 times.

It would be great to have a piece of subordinate code that could compute the 5th power of *anything*. Of course, there would have to be a way to tell the subordinate code what number to work with. The following application does just that, using methods:

```
1. public class UsesMethods
2. {
3.     public static void main(String[] args)
4.     {
5.         int n = 5;
6.         int n5th = toThe5th(n);
7.         System.out.println(n + " >=> " + n5th);
8.         n = 6;
9.         n5th = toThe5th(n);
10.        System.out.println(n + " >=> " + n5th);
11.        n = 7;
12.        n5th = toThe5th(n);
13.        System.out.println(n + " >=> " + n5th);
14.        n = 8;
15.        n5th = toThe5th(n);
16.        System.out.println(n + " >=> " + n5th);
17.        n = 9;
18.        n5th = toThe5th(n);
19.        System.out.println(n + " >=> " + n5th);
20.    }
21.
22.    static int toThe5th(int x)
23.    {
24.        int result = x * x * x * x * x;
25.        return result;
26.    }
27. }
```

The application's output is the same as the output from the previous version.

The code from lines 22-26 constitutes a method. Line 22 is called the method's *declaration*. It tells the compiler that what is about to follow will be the definition of the method whose name (along with some other information) appears in the declaration line. The definition, or *body*, of a method immediately follows the declaration, and it must appear within curly brackets.

The general format of a method declaration is

Optional_modifiers Return_type Name(Optional_arguments)

The only mandatory parts of a declaration are the return type, the name, and the parentheses. In this example, we have one modifier (static), the return type is int, the method's name is toThe5th, and there is one argument (int x) that appears inside the parentheses. Let's look at each of these elements.

You have already been patiently tolerating the unexplained presence of the *static* modifier in every application we have looked at in this book. It has appeared in the declaration of `main`, which is a method that appears in every Java application.

Understanding `static` will become much easier after we introduce object-oriented programming in [Chapter 7](#). For now, let's just say that `static` means "don't be object-oriented." Other modifiers that might appear in a method declaration include access modifiers, which will be presented in [Chapter 9](#).

We'll look at the return type in a moment. Let's move now to the method name. The rules for the name are the same as those for variable names: The first character must be a letter, an underscore, or a dollar sign. The subsequent characters may be any of these, or they may be digits.

As with variable names, you have a broad choice. You should pick the name that does the best possible job of describing what the method does. When the name appears outside the method, as in lines 6, 9, 12, 15, and 18 of this example, the path of program execution detours through the method. This is known as *calling* or *invoking* the method, and a line that calls a method is the method's *caller*.

Note that in the lines where the method is called, the method call (the name followed by something parenthetical) is used in a context where you would expect to see a value. Until now, the right-hand side of an assignment has been either a literal, a variable, or an arithmetic or boolean expression composed of literals, variables, and operators. Now we add something new to the mix. Anywhere the compiler expects a value, you can use a method call. This is because a method call produces a value, called the method's *return value*. When the computer executes a line of code that includes a method call, the computer takes a detour through the method body in order to compute the return value. When the detour is finished, execution continues where it left off.

The arguments are the method's inputs. In this example, the argument list is `int x`. This means that the method has one input, whose type is `int`. Within the body of the method, that input will be called `x`. When the method runs, the actual value of `x` will be whatever the caller wants. The caller specifies an input value by putting the value in parentheses in the call line. Lines 6, 9, 12, 15, and 18 all pass `n` as the method argument, but the value of `n` is different for each of those lines. In line 6, `n` is 5, so the call from line 6 will execute with `x` set to 5. In line 9, `n` is 6, so the call from line 9 will execute with `x` set to 6. And so on. This demonstrates the flexibility of methods.

The return type in the method declaration tells the type of the return value. The returning of a value happens in line 25, where we see the `return` keyword. This causes the path of execution to return to the caller line. The value following `return` is the return value. In this example, the return value is `result`, which is the 5th power of the argument.

Argument Lists

The `toThe5th` method in the previous example took a single argument, so within the parentheses in the method declaration, we saw a single type (`int`) followed by a single name (`x`). You can create methods with an arbitrarily number of arguments of arbitrary types. To do this, just write a declaration with the following format:

```
Mods Ret_type Name(type0 arg0, type1 arg1, type2 arg2 ...)
```

Note Note that the numbering of the arguments starts at 0, rather than 1. Whenever you see someone do this, you can be certain that they work in the computer field, where counting from 0 is conventional. We already saw this in [Chapter 1](#), where SimCom's memory addresses started at 0. We will see it again in [Chapter 6, "Arrays."](#) It is important to get into the habit of counting from 0 as soon as possible, even though there is a slight inconvenience. When you start counting from 1, the last number you count out is the actual number of things you've counted. When you start from 0, the actual number is 1 more than the last number you've counted out. So in the preceding declaration format, we have 3 arguments, and the largest index is 2.

The following method takes 2 arguments:

```
static double hypotSquared(double leg0, double leg1)
{
    return leg0*leg0 + leg1*leg1;
}
```

Recall that in any arithmetic expression, multiplication takes precedence over addition, so the method really does return the square of the hypotenuse.

The MethodLab animated illustration demonstrates the passing of arguments and the returning of return values. To run the program, type `java methods.MethodLab`. The display presents code that calls a method, as shown in [Figure 4.1](#).

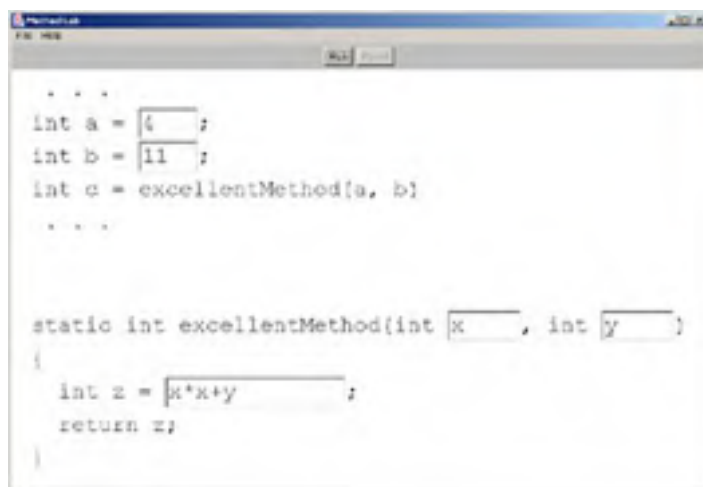


Figure 4.1: MethodLab

Click on the Run button to see the animation. The black lines indicate the passing of arguments, which are called *a* and *b* in the caller but *x* and *y* in the method. The blue line represents returning the return value. When the animation is finished, click Reset to start again. [Figure 4.2](#) shows MethodLab after the animation finishes.

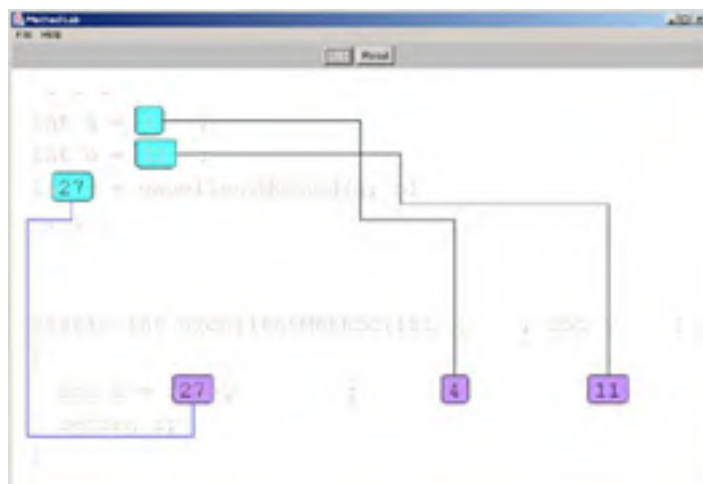


Figure 4.2: MethodLab after animating

You can customize MethodLab by typing any integer value you like into the text fields for *a* and *b*. You can also enter any numeric formula for the value of *z*, which becomes the return value. Try the formula in the preceding example. Since this method's arguments are called *x* and *y*, the formula should be

```
x*x + y*y
```

With this formula, try running MethodLab with *a* = 3 and *b* = 4. Try again with *a* = 12 and *b* = 5. These combinations are the only small integers that represent triangles with integer hypotenuses.

A method's argument list can be arbitrarily long. At times you might even want a method with no arguments at all. In that case, the method's declaration has an empty pair of parentheses after the name, and the caller passes nothing at all inside its own parentheses:

```
float f = sayHello();  
...  
static float sayHello()  
{  
    System.out.println("Hello");  
    return 3.14159f;  
}
```

The important thing is that when you call a method, the call should have the same number of arguments as the method declaration, and the types of the arguments passed by the caller should be compatible with the types in the method declaration. This is subtly different from saying that the caller's argument types should exactly match the types in the method declaration. Recall from [Chapter 3](#) that a value can be assigned to a variable of a different type, provided the new type is wider than the old type. [Figure 4.3](#) (first shown in [Chapter 3](#)) shows the width relationships among the numeric primitive types.

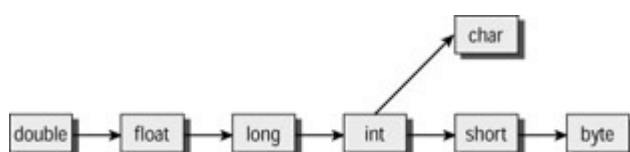


Figure 4.3: Numeric type widths

The rule for passing method arguments is similar to the rule for assignment: *You can pass an argument whose type is different from the type declared by the method, provided the type declared by the method is wider than the type that you pass.* Recall that a type is wider than another type if you can get from the first type to the second type by following the arrows in [Figure 4.3](#).

For example, suppose a method has the following declaration:

```
static char abcde(long wayToGo)
```

This method can be called with a long argument, or with any argument whose type is narrower than long: byte, short, char, or int.

More on Return Types

At times, you might want to create a method that doesn't return anything. The method might print out a message, display a dialog box, or store a value in a file. In cases such as these, it is difficult to think of any value that the method could meaningfully return, and concocting a return value just for the sake of having one would not contribute to the quality of the program. (A basic principle of writing fiction is that every word should contribute to developing the plot or developing the characters. We can invent a similar principle for writing software: Every word of source code should contribute to the operation or the readability of the program.)

Suppose you want a method that prints a number along with a message. Since no return value is needed or relevant, replace the return type in the declaration with the word `void`:

```
static void printPretty(int x)
{
    System.out.println("x = " + x);
}
```

Note the absence of a `return` statement. A void method runs until it executes its last line. Then it returns automatically. Optionally, you can add a `return` statement with no value at the end of the method:

```
static void printPretty(int x)
{
    System.out.println("x = " + x);
    return;
}
```

Here the `return` statement doesn't contribute to program execution or readability (we already know that the method returns when it hits bottom), but later we will see cases where explicitly saying `return` can be useful.

A call to a method with a non-void return type can be used anywhere that a variable or literal of the same type can be used: as the right-hand side of an assignment, as an operand in an operation, or even as an argument of another method call. By contrast, a call to a void method has no type. So if method `iAmVoid` is void, you could not say `int z = iAmVoid();` because there would be no value to assign to `z`. When you call a void method, you just want it to do its thing, so you make the call all by itself, followed by a semicolon, like this:

```
iAmVoid();
```

If a method changes the state of the program or the computer in any way (other than returning the return value), the change is called a *side effect*. Clearly, when you call a void method, you do so because you are interested in a side effect. (In this example, the side effect was the printing of the message.) Sometimes you might want to call a non-void method, not because you are interested in the return value, but because you want the side effect. In that case, you can just call the method as if it were void.

For example, the following method both prints out and returns hypotenuse squared:

```
static int vocalHypotSquared(int a, int b)
{
    int hSquared = a*a + b*b;
    System.out.println("h-squared = " + hSquared);
    return hSquared;
}
```

If you just wanted to print out the message, you could call the method like this, ignoring the return value:

```
vocalHypotSquared(5, 12);
```

Polymorphism

Polymorphism comes from the Greek for "many forms." It is one of several five-syllable words pertaining to object-oriented programming. In our context, it means that a method can have one name but many forms. In other words, you can define multiple methods with the same name.

At first, this might seem impossible. How can the system know which of the various methods you had in mind? The rule is that if two methods have the same name, their argument lists have to be different. That is, the types that appear in lists must differ; the argument names are not considered here. If a method name appears more than once, we say that the name is *overloaded*.

For example, the following two methods could appear in the same program:

```
static int getMass(int n)
{
    ...
}
static int getMass(double a, char c)
{
    ...
}
```

Here, `getMass()` is legitimately overloaded. However, the following two methods could *not* appear in the same program:

```
static int getMass(int n)
{
    ...
}
static int getMass(int x)
{
    ...
}
```

The argument names are different, but that doesn't help. Both method versions have the same name, and each version takes a single argument of type `int`, so the compilation will fail. If the argument types were different (for example, if the argument of `getMass()` had type `long` or `byte`), the code would compile without error.

Methods That Call Methods

Methods can be called from anywhere – even from other methods. In fact, any complicated program is likely to consist of methods that call other methods that call other methods, and so on, to many levels of depth. For example, you might have a method that prints out two values:

```
static void print2Vals(int val0, int val1)
{
    System.out.println(val0 + " and " + val1);
}
```

Now what if you want a method that prints out the cubes of its two arguments? You might do it as follows:

```
static void print2Cubes(int val0, int val1)
{
    int val0Cubed = val0*val0*val0;
    int val1Cubed = val1*val1*val1;
    System.out.println(val0Cubed + " and " + val1Cubed);
}
```

Since you already have the `print2Vals` method, you can rewrite `print2Cubes` as follows:

```
static void print2Cubes(int val0, int val1)
{
    int val0Cubed = val0*val0*val0;
    int val1Cubed = val1*val1*val1;
    print2Vals(val0Cubed, val1Cubed);
}
```

Now you have a method, (`print2Cubes`) that calls another method (`print2Vals`). You can rewrite `print2Cubes` to be even more terse, as follows:

```
static void print2Cubes(int val0, int val1)
{
    print2Vals(val0*val0*val0, val1*val1*val1);
}
```

Since the multiplication is repetitious, you can also introduce a new method:

```
static int nCubed(int n)
{
    return n*n*n;
}
```

Now `print2Cubes` becomes

```
static void print2Cubes(int val0, int val1)
{
    print2Vals(nCubed(val0), nCubed(val1));
}
```

Now you have a method, (`print2Cubes`) that consists of a single call to another method (`print2Vals`), and that call's arguments are both expressed as method calls (to `nCubed`). This kind of structure—calls to calls to calls—is perfectly typical of programs.

Passing by Value

In formal computer terminology, we say that Java passes *by value*. This is just another way to say that methods get copies of their arguments and have no access to the original values. The alternative is called passing *by reference*, where methods work with the caller's data and not with copies. The latter is a perfectly valid way to design a language; it's just not the way Java does it.

When a caller calls a method and the flow detours into the method's body, the JVM copies the argument values provided by the caller and gives the copies to the method. This means that if the method alters its arguments, the alteration has no effect on the caller.

Consider the following method:

```
static void print3x(int x)
{
    x = 3*x;
    System.out.println("3 times x = " + x);
}
```

The method triples its argument. You might wonder what happens to the value that the caller passes to the method. For example, what does the following print out?

```
int z = 10;
print3x(z);
System.out.println("Now z is " + z);
```

Does this code print "Now z is 10" or "Now z is 30"? If the method has access to the actual data passed in by the caller, the code should print out "Now z is 30". However, the code actually prints "Now z is 10" because the method triples its own private copy, not touching the

caller's version. After the method returns, the memory used for storing the method's copy is recycled. The method's copy does not survive after the method returns.

Order

A program is, in large part, a collection of methods that call other methods. These other methods call still other methods, and so on.

Within an application, methods can appear in any order. Once again, you are in a situation where your choices can make a program either easier or harder for others to read. It makes sense to put related methods near one another. It is common practice, though by no means universal, to put the `main` method at the very end, just before the final closing curly bracket. From here on, this book will follow that convention.

Scope

When you write a method declaration, you can choose almost any argument names you like. Of course, the names have to be legal (beginning with a letter, underscore, or dollar sign, and continuing with the same plus digits). Moreover, the name should be indicative of the argument's meaning. But beyond these considerations, you have complete latitude. In particular, you are allowed to reuse a variable name that has been used elsewhere in your program.

Every Java variable has a *scope*. A variable's scope is the matching pair of open and closed curly brackets that most tightly encloses the variable's declaration. Another way to say this is in terms of blocks. A *block* is a contiguous piece of code that begins with an open curly and ends with a matching closed curly. Blocks may contain many kinds of code. We have already seen blocks that contain method bodies. Later in this book, we will see blocks that contain, among other things, other blocks. Those inner blocks can contain, among other things, still other blocks, and so on, to whatever depth is useful.

Already we have seen blocks that contain other blocks, since every Java application is a block that looks like this:

```
1. public class ClassName
2. {
3.     // Optional other methods.
4.     public static void main(String[] args)
5.     {
6.         ...
7.     }
8. }
```

Any variable defined in the main method has a scope that spans from line 5 through line 7, since that is the tightest matched pair of curlys that would contain the variable's declaration.

The scope of a method argument is the method itself, even though the argument is actually declared just before the open curly that begins the scope.

Now, here is why it is so important to know about scope: A variable name may not be declared more than once in a single scope. However, a name that is declared in one scope may be declared and used in any number of other scopes. Each declaration refers to a different variable; the variables just happen to have the same name. The situation is similar to filenames in directories. Names must be unique within any particular directory, but a filename that appears in one directory may be used in another directory. The two files have nothing to do with each other, and the common name is just a coincidence.

Consider the following example:

```
1. public class ReusesNames
2. {
3.     static void printTriple(int x)
4.     {
5.         int i = 3*x;
6.         System.out.println("Triple = " + i);
7.     }
8.
9.     public static void main(String[] args)
10.    {
11.        int x = 10;
12.        int i = x+5;
13.        printTriple(i);
14.    }
15. }
```

Here we have two methods, `main` and `printTriple`, each with its own scope. Each method's scope has its own `i` and its own `x`, unrelated to the `i` and `x` of the other method. Each method can use and modify its own `i` and `x`, but cannot touch the `i` and `x` of the other method.

A convenient effect of Java's scoping rule is that when you write a method, you don't have to worry about whether a variable name you like is already in use in a different method. This is especially convenient in a long program that might have hundreds of methods, each with a dozen variables. If it were not for the scoping rule, we would quickly run out of good variable names.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Which of the following are legal method names?

1. \$25
2. 25\$
3. abc_
4. _ABc

2. Suppose you want to write a method that returns the diameter of a planet, in millimeters. Since it's your program, you can choose any name you like for the method. Rank the following method names, from worst to best. Use your own judgment as to what makes one method name better or worse than another.

1. getPlanetDiameter
2. getSize
3. getPlanetDiameterMm
4. getIt
5. getPlanetSize

3. Suppose a method has the following declaration:

```
static int abc(int x, short y)
```

Suppose this method is called as follows:

```
abc(first, second)
```

Which of the following are legal types for the variables `first` and `second`?

1. `int first, int second`
2. `short first, short second`
3. `byte first, char second`
4. `char first, byte second`

4. Consider the following method declaration:

```
xyz(double d)
```

Which argument types can a caller pass into this method?

5. Earlier in this chapter, you learned that if method `iAmVoid` is void, you can't say `int z = iAmVoid();` because there is no value to assign to `z`. What happens if you try? Write a program that does this experiment.

6. Earlier in this chapter, you saw the following method:

```
static void print3x(int x)
{
    x = 3*x;
    System.out.println("3 times x = " + x);
}
```

The following code prints out "Now z is 10", not "Now z is 30", because the method modifies its own private copy of the argument:

```
int z = 10;
print3x(z);
System.out.println("Now z is " + z);
```

Write a program that proves this.

Chapter 5: Conditionals and Loops

The [previous chapter](#) showed you how method calls can be used to detour the flow of program execution. This chapter will introduce two more ways to vary program flow: conditionals and loops. By the end of this chapter, you will be able to write programs in which control flows in quite intricate patterns.

Conditionals

If a method call is like a detour in the path of execution, then a *conditional* is like a fork in the road. Conditional code is executed only if a certain criterion is met, typically when a certain boolean expression evaluates to `true`.

We will begin with the `if` statement, which is Java's most basic conditional. We will also look at the more complicated ternary operator and `switch` statement.

if

In its simplest form, the `if` statement looks like this:

```
if (boolean_expression)
    do_something;
```

The code immediately following the `if` keyword must be of boolean type and must be enclosed in parentheses. The code that follows the parenthetical boolean expression can be either a single statement or a block of statements enclosed in curly brackets. Let's look at some examples.

The following code fragment prints out a message if `x` is divisible by 10:

```
if (x%10 == 0)
    System.out.println("x is divisible by 10.");
```

In the next example, `y` and `z` are both reduced if their product exceeds 1,000:

```
if (y*z > 1000)
{
    y -= 10;
    z -= 20;
}
```

Note in the previous example that if the condition is met, the action to be taken consists of two statements. When the conditional action is longer than a single statement, the multiple statements of the action are enclosed in curly brackets.

if and else

An `if` statement can be enhanced with the `else` keyword. You can only use `else` after the statement or curly bracket-enclosed block that follows an `if`. As with `if`, the code that follows `else` can be either a single statement or a block of statements within curly brackets. As you might expect, the code following `else` is executed if the `if` statement's boolean expression evaluates to `false`.

For example, the following code prints out a message that depends on whether the value of `x` is even or odd:

```
if (x%2 == 0)
    System.out.println(x + " is even.");
else
    System.out.println(x + " is odd.");
```

In the next example, the method "clamps" the value of its `z` argument. The return value is `z`, unless `z` exceeds a lower or upper limit. If this is the case, the return value is the exceeded limit:

```
static long clamp(long z, long lowLimit, long highLimit)
{
    if (z < lowLimit)
        return lowLimit;
    else if (z > highLimit)
        return highLimit;
    return z;
}
```

If curly bracket-enclosed code blocks are used after `if` or `else`, those blocks themselves can contain `if` statements. The following code fragment uses *nested* `if` statements:

```
if (x > 1000000)
{
    // x is big.
    if (x%2 == 0)
        System.out.println("Big and even.");
    else
        System.out.println("Big and odd.");
}
else
{
    // x is little.
    if (x%2 == 0)
        System.out.println("Small and even.");
}
```

```
else
    System.out.println("Small and odd.");
}
```

Note There is no limit to how deeply you can nest `if` statements. Of course, if you nest too deeply, your code becomes difficult to read and understand. See the "[Nesting](#)" section later in this chapter for an explanation of this technique.

else if

In the [previous section](#), you learned how to follow an `if` statement with an `else` statement. You can also follow an `if` statement with an arbitrary number of `else if` statements. An `else if` statement is like an `else` statement, but it is followed by a parenthetical boolean expression and then by a single statement or curly bracket-enclosed block. As you might expect, the single statement or curly bracket-enclosed block is executed only if the boolean expression evaluates to `true`. There is no limit to the number of `else if` statements that may follow an `if` statement, and the last `else if` statement may be followed by an `else` statement.

The following example is a method that prints out one of a number of possible messages, based on the size of the `z` argument:

```
static void howBig(double z)
{
    if (z < 0.001)
        System.out.println("Very tiny");
    else if (z < 1)
        System.out.println("Tiny");
    else if (z < 100)
        System.out.println("Medium");
    else if (z < 100000)
        System.out.println("Large");
    else
        System.out.println("Very large");
}
```

Note that the series of tests on the value of `z` begins with a straightforward `if` statement, followed by three `else if` statements. The `else` statement comes at the end, which is the only place where it may appear.

Before we continue, let's take a moment to appreciate the power of the various versions of the `if` statement. The Java functionality presented in the previous chapters of this book, while impressive, amounts to using your computer as a very fast calculator. For instance, a method would always process its arguments in exactly the same way. With the introduction of `if` statements, we have programs that can react flexibly. The `howBig` method, for example, can react flexibly to the value of its `z` argument.

Later in this chapter we will examine loops, which introduce an additional level of flexibility. But first, let's look at two more kinds of conditional execution: the ternary operator and the `switch` statement.

The Ternary Operator

In [Chapter 3, "Operations,"](#) we looked at Java's unary and binary operators. Now let's look at the *ternary* operator. The name *ternary* just means that there are three operands. Since there are three, we will need two symbols to separate them: the question mark (?) and the colon (:). The operator is used like this:

```
boolean_expression ? value_1 : value_2
```

The value of the ternary operation depends on the value of the boolean expression. If the boolean expression evaluates to `true`, the value of the overall operation is `value_1`. If the boolean expression evaluates to `false`, the value of the overall operation is `value_2`.

Typically, a ternary operation appears on the right-hand side of an assignment. For example, suppose you want `radius` to be 10 if `mass` is less than or equal to 50,000; otherwise, you want `radius` to be 99. Without the ternary operator, you could do it this way:

```
if (mass <= 50000)
    radius = 10;
else
    radius = 99;
```

You can rewrite this in a single line with the ternary operator:

```
radius = mass <= 50000 ? 10 : 99;
```

The boolean expression does not need to appear in parentheses, but the line is more readable like this:

```
radius = (mass <= 50000) ? 10 : 99;
```

The ternary operator is a convenient replacement for an `if...else` expression.

Now let's look at the `switch` statement, which is a convenient replacement for a sequence of `if...else` expressions.

switch

In the [previous section](#), you saw how the ternary operator can replace certain `if...else` structures. We will now look at the `switch` statement, which can replace entire chains of `if...else if...else if` structures.

Suppose you wanted to write some code that takes special actions if the value of a `char` called `theChar` is a vowel (a, e, i, o, or u). The special actions consist of printing a message and setting the value of an `int` called `vowelNum`. Using `if` and `else`, you could write the code as follows:

```
if (theChar == 'a')
{
    System.out.println("a is a vowel.");
    vowelNum = 0;
}
else if (theChar == 'e')
{
    System.out.println("e is a vowel.");
    vowelNum = 1;
}
else if (theChar == 'i')
{
    System.out.println("i is a vowel.");
    vowelNum = 2;
}
else if (theChar == 'o')
{
    System.out.println("o is a vowel.");
    vowelNum = 3;
}
else if (theChar == 'u')
{
    System.out.println("u is a vowel.");
    vowelNum = 4;
}
}
```

This can be rewritten as follows, using a `switch` statement:

```
switch (theChar)
{
    case 'a':
        System.out.println("a is a vowel.");
        vowelNum = 0;
        break;
    case 'e':
        System.out.println("e is a vowel.");
        vowelNum = 1;
        break;
    case 'i':
        System.out.println("i is a vowel.");
        vowelNum = 2;
        break;
    case 'o':
        System.out.println("o is a vowel.");
        vowelNum = 3;
        break;
    case 'u':
        System.out.println("u is a vowel.");
        vowelNum = 4;
        break;
}
```

The value in parentheses just after the `switch` keyword is called the *expression* of the `switch` statement, and it must be of type `byte`, `short`, `char`, or `int`. (This example assumes that `theChar` has been declared to be a `char`.) When the `switch` code is executed, Java searches through the `case` statements, looking for one that matches the expression's value. If no match is found, nothing happens; execution continues after the closing curly bracket. If a match is found, control jumps to the first executable line following the `case` statement. Then execution proceeds line by line until a `break` statement is reached. At this point, execution of the `switch` code is terminated, and control continues after the closing curly bracket.

switch and default

The keyword `default`, followed by a colon, can appear in place of a `case` statement. The code following the `default` statement is executed if none of the `case` statements match the expression. For example, suppose you want to modify your code so that it prints out "Not a vowel" if `theChar` is not a vowel. If you couldn't use a `switch` statement, you would do the following:

```
if (theChar == 'a')
{
    System.out.println("a is a vowel.");
    vowelNum = 0;
}
else if (theChar == 'e')
{
    System.out.println("e is a vowel.");
    vowelNum = 1;
}
else if (theChar == 'i')
{
    System.out.println("i is a vowel.");
    vowelNum = 2;
}
else if (theChar == 'o')
{
    System.out.println("o is a vowel.");
    vowelNum = 3;
}
```

```
}
else if (theChar == 'u')
{
    System.out.println("u is a vowel.");
    vowelNum = 4;
}
else
    System.out.println("Not a vowel.");
```

This code is the same as the original solution, but with a final `else` at the end. The following code uses a `switch` statement with a default block to achieve the same result:

```
switch (theChar)
{
    case 'a':
        System.out.println("a is a vowel.");
        vowelNum = 0;
        break;
    case 'e':
        System.out.println("e is a vowel.");
        vowelNum = 1;
        break;
    case 'i':
        System.out.println("i is a vowel.");
        vowelNum = 2;
        break;
    case 'o':
        System.out.println("o is a vowel.");
        vowelNum = 3;
        break;
    case 'u':
        System.out.println("u is a vowel.");
        vowelNum = 4;
        break;
    default:
        System.out.println("Not a vowel.");
        break;
}
```

When you look at all four versions of this example, you can see that using a `switch` statement does not significantly reduce the number of lines of code (although there is a reduction). The main benefit is readability. The `switch` versions more clearly tell readers what is happening.

Omitting the *break*

Once a `case` block is found that matches the `switch` statement's expression, execution continues until a `break` is reached or the `switch` statement's closing curly bracket is reached, whichever comes first. If a `case` block does not end with a `break`, execution continues past the next `case` statement and into the code for that `case` block.

In the previous example, suppose the `case` block for 'e' did not end with a `break` statement. (Perhaps due to an innocent oversight. It's only human to forget to type `break` from time to time.) The code would then look like this:

```
. . .
7. case 'e':
8.     System.out.println("e is a vowel.");
9.     vowelNum = 1;
10. case 'i':
11.     System.out.println("i is a vowel.");
12.     vowelNum = 2;
13.     break;
. . .
```

We've added line numbers for easy reference. The `switch` statement detects that the expression value ('e') matches the case on line 7. The message on line 8 is printed out, and then at line 9 `vowelNum` is set to 1. Since there is no `break` at line 10, execution just keeps on going. The `case` statement at line 10 is ignored, and control flow continues at line 11. The message on line 11 is printed out, and at line 12 `vowelNum` is set to 2. At last we have a `break`, so execution of the `switch` is finished.

This behavior of continuing from one case to the next in the absence of a `break` statement is called *falling through*. Falling through is a mixed blessing. When it happens because you forgot to type `break` for a particular case, it's just a bug that might be hard to find (but easy to fix once it's found).

On the other hand, falling through might be just the behavior that you want. The feature is especially useful when you want to use the same code to process more than one case. The letters `y` and `w` are sometimes considered to be vowels. (*Y* occasionally, as in *occasionally*; *w* very rarely, as in *crwth*, a medieval musical instrument, pronounced "crooth.") You might want to print out a special message if `theChar` has either of these values. If you were using `if...else` code, you would insert the following lines:

```
. . .
else if (theChar == 'y' || theChar == 'w')
    System.out.println("y and w are sometimes vowels.");
. . .
```

You can incorporate this test into your `switch` code by inserting the following lines:

```
. . .
case 'y':
case 'w':
    System.out.println("y and w are sometimes vowels.");
    break;
. . .
```

Where should these lines be inserted? Strictly speaking, the cases in a `switch` statement, including the `default` code, can appear in any order. However, for readability, it makes the most sense to have the cases appear in their natural order (numerical or alphabetical), with the `default` code appearing last.

Now that we have looked at Java's conditional code, we can turn our attention to loops.

Team LiB

← PREVIOUS NEXT →

Loops

You have seen that conditional code is like a fork in the path of program execution. Extending this analogy, a loop is like an eddy or a whirlpool. No, wait, that can't be right... paths don't have whirlpools. The analogy has broken down. At any rate, a loop is a piece of code that is executed repeatedly. The number of repetitions can be some preset value, or the loop can run on and on until a condition is met.

We will begin with `while` loops, and then move on to `for` loops. We will also look at several techniques for enhancing loop behavior: breaking, continuing, and nesting.

While Loops

A while loop is a chunk of code that is executed repeatedly until a certain condition is met. The format for a while loop is

```
while (expression)
    loop_body
```

The expression must be of the boolean type. The loop body is the code to be repeated. This can be either a single statement or a block of code enclosed in curly brackets. Initially the expression is evaluated, and if its value is `true`, the loop body is executed once. Then the expression is evaluated again, and if its value is still `true`, the loop body is executed once again. This happens again and again and again. Eventually (we hope), the expression evaluates to `false`. When this happens, the loop body is not executed anymore. Instead, control jumps to the code immediately after the loop body.

This explanation might seem paradoxical. If the while loop is to be of any use, the expression must initially evaluate to `true`. Otherwise, the loop body won't be executed at all. But if the expression is indeed initially `true`, how can the loop ever terminate?

The answer is that either the expression or the loop body must modify the data from which the expression is calculated. Let's look at a few examples of how this works.

First, here is a useless loop that prints too many messages:

```
int x = 23;
while (x > 0)
    System.out.println("Still going!");
```

This loop runs forever or until you press Ctrl+C to terminate the program, whichever comes first. This example demonstrates the need to somehow modify the data that constitutes the expression.

The next example is more useful. The following code prints the numbers 1 through 10:

```
int counter = 1;
while (counter <= 10)
{
    System.out.println("counter = " + counter);
    counter += 1;
}
```

You can use a pre-increment or post-increment operator to make this code slightly more terse:

```
int counter = 1;
while (counter <= 10)
{
    System.out.println("counter = " + counter);
    counter++;
}
```

The next example prints out consecutive square numbers that are less than 1,000:

```
int counter = 1;
while (counter*counter < 1000)
{
    System.out.println(counter * counter);
    counter++;
}
```

This example points out a useful feature of while loops: You don't need to know beforehand how many passes you want to make through the loop body. You could certainly find a calculator, figure out that the square root of 1,000 is 31.6227766..., and write the following:

```
int counter = 1;
while (counter <= 31)
{
    System.out.println(counter * counter);
    counter++;
}
```

This approach works, but it violates the spirit of making the computer compute. If you're programming a computer, you shouldn't have to reach for a calculator. With the next-to-last version of the example, you take advantage of the fact that you don't have to know how many passes you're going to make through a while loop. You just have to be able to know when you're done.

While loops are the first of several kinds of loops that we will present in this chapter. Loops are powerful because a few lines of source code can cause the computer to execute a very large number of instructions.

The WhileLab animated illustration demonstrates while loops. To run the program, type `java loops.WhileLab`. You see a display that shows a while loop with two assignment lines in its body, as shown in [Figure 5.1](#).

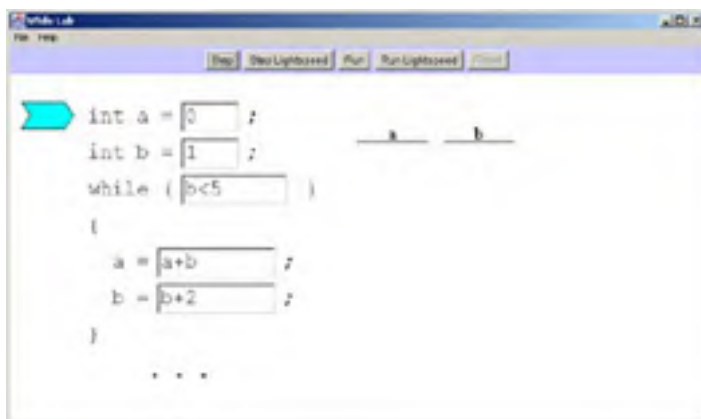


Figure 5.1: While Lab: initial display

The loop uses variables `a` and `b`. As the code executes, their values are displayed and updated. Click on the Step button to animate the next line of code. Click on the Run button to animate the entire loop. You can click on Step Lightspeed or Run Lightspeed to bypass the animation and just see the result. When the animation is finished, click Reset to start again.

You can type in your own values for the initial values of `a` and `b`, for the test expression, and for the new values that are assigned to `a` and `b` within the loop. Figure 5.2 shows While Lab with a slightly modified test expression.

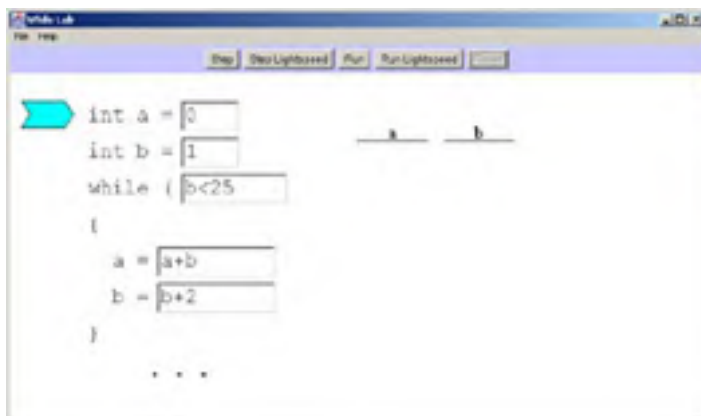


Figure 5.2: While Lab with modified test expression

Figure 5.3 shows the result of executing the configuration shown in Figure 5.2.

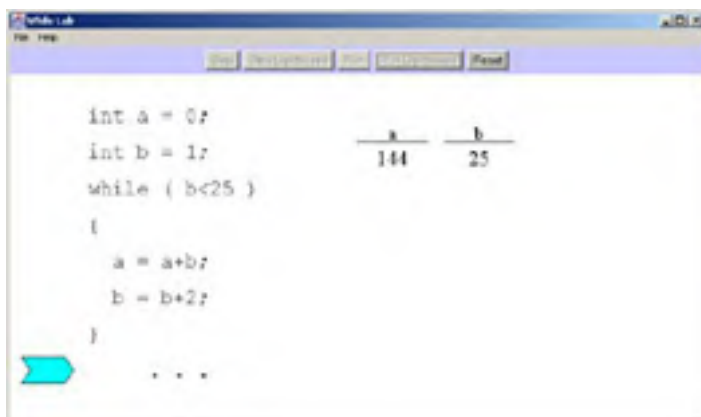


Figure 5.3: While Lab after execution

Try typing in different values for `b <= ??` in the test expression. (If you enter a large number, the loop will be executed a large number of times, so you probably want to execute with the Run Lightspeed button rather than the Run button.) As you vary the limit on `b`, what do you notice about the final value of `a`?

Try configuring WhileLab's code display so that the code computes the following results (which can be in either `a` or `b`, whichever you prefer):

- The sum of the numbers 1 through 500, inclusive.
- The sum of the even numbers from 50 through 60, inclusive.
- The product of the first five odd numbers.

Now that you have some experience with while loops, we can look at a variation on the theme: do-while loops.

Do-While Loops

A while loop always tests its condition before executing its body. There may be times when you want to execute the body first, and then test. This is done with a do-while loop. The format of a do-while loop is

```
do
    loop_body
while (expression);
```

As with ordinary while loops, the loop body can be either a single statement or a curly bracket-enclosed block. Note that the parenthetical expression must be followed by a semicolon.

When a do-while loop is executed, the loop body is executed. Then the expression is evaluated. If the expression evaluates to `true`, the loop body is executed again, the expression is evaluated again, and so on until eventually the expression value is `false`. At that point, execution of the loop is finished. As with ordinary while loops, you should write do-while code in such a way that during execution of the loop, the data constituting the expression changes so that at some point the expression's value can become `false`.

The code in the following example prints out the cube of `x`, and then increments `x` by 5, until `x` exceeds 100:

```
do
{
    System.out.println(x*x*x);
    x += 5;
}
while (x <= 100);
```

The body of a do-while loop is always executed at least once. In the preceding example, at least one line will be printed out, even if the initial value of `x` is greater than 100.

Do-while loops are not very different from while loops. The main difference is that the body of a while loop might not ever be executed, whereas the body of a do-while loop will always be executed at least once.

Now let's look at for loops, which are useful when you know how many passes through the loop body you want.

For Loops

The following code, which uses a while loop to compute a value and print a message ten times, has a very common structure:

```
int z = 0;
while (z < 10)
{
    int formula = z*z*z + z*z;
    System.out.println(formula);
    z++;
}
```

The code first initializes `z`, and then it enters a while loop. Within the loop body, the first two lines perform the internal business of the loop, so to speak. The last line (`z++`) is concerned with updating the only data that changes from pass to pass in the loop. [Figure 5.4](#) shows the structure of the loop.

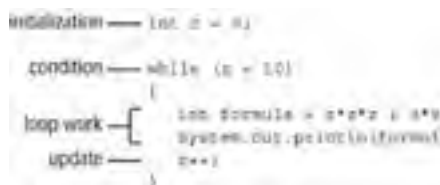


Figure 5.4: A common loop usage

This structure (initializing before a loop, incrementing at the end of the loop body) is so common that there is a special kind of loop to support it. The `for` loop has the following format:

```
for (initialization; condition; update)
    body
```

This `for` loop is exactly equivalent to

```
initialization;
while (condition)
{
    body
    update
}
```

The `for` keyword must be followed by three items, known as the *initialization*, the *condition*, and the *update*. These are enclosed in parentheses and separated by semicolons. When the `for` loop is processed, the initialization is executed once. Then the condition, whose type must be boolean, is evaluated. If the condition is `true`, the loop body is executed. Then the condition is evaluated again, and so on until the condition is `false`. Notice that no matter how many times the loop body is executed (zero times, once, or multiple times) the initialization is executed exactly once.

`For` loops are useful when you know beforehand how many times you want the loop body to be executed. (They are especially useful when you're processing arrays, which will be presented in the [next chapter](#).) When you don't know beforehand how many times the body should execute, you are generally better off using a while loop because your code will be less complicated.

Usually, the initialization involves setting the value of a single variable, often to zero. The condition is usually a test on the value of that variable, and the update increments the variable. For example, the following code prints out a message 10 times:

```
int i;
for (i=0; i<10; i++)
    System.out.println("DANGER!");
```

In this code, the variable `i` is used just to regulate the number of passes through the loop body. Since it does not appear in the body, we could have chosen any name for the variable, but it is conventional to use `i` (or `j` if `i` is in use). A variable used in this way (regulating the number of passes through the loop body, but otherwise playing little or no role in the body) is called a *loop counter*. We could have initialized `i` to any value, as long as the value in the condition was 10 greater than that, but it is conventional to start a loop counter at 0. If you follow these conventions, and we strongly recommend that you do so, people who read your code will have a good chance of understanding your intentions.

The initialization and update portions of a for loop can have multiple parts, separated by commas. For example, the following code prints out the areas of rectangles whose bases range from 5 to 10 inches, and whose heights are 2 inches more than the base:

```
int base, height;
for (base=5, height=7; base<=10; base++, height++)
{
    int area = base * height;
    System.out.println(area + " square inches");
}
```

Here, both the initialization and the update have multiple parts.

Breaking and Continuing

Usually a loop runs until its condition is `false`. However, there may be times when you want to terminate the loop prematurely. This is called *breaking out of the loop*.

As an example of loop breaking, imagine you are writing a payroll program for a small company. The company has 100 employees whose ID numbers are 1001 through 1100. A method called `getPayAmount`, which takes an employee ID as its argument, returns the amount of money the employee should be paid. Another method called `printCheck`, which has an employee ID and an amount as its arguments, prints the specified employee's paycheck. A variable called `balance` keeps track of how much money the company has in the bank.

The following code prints everybody's paycheck and keeps track of the bank balance:

```
int id;
for (id=1001; id<=1100; id++)
{
    float pay = getPayAmount(id);
    printCheck(id, pay);
    balance -= pay;
}
```

The problem with this code is that it doesn't take precautions against using up all the money in the bank account. The following code uses a `break` statement to terminate the loop as soon as there isn't enough money left to cover the next paycheck:

```
int id;
for (id=1001; id<=1100; id++)
{
    float pay = getPayAmount(id);
    if (balance-pay < 0)
        break;
    printCheck(id, pay);
    balance -= pay;
}
```

The `break` statement causes immediate termination of the loop. Execution continues with the first line of code following the loop. You can break out of any kind of loop: `do`, `do-while`, and `for`. Note that this application of `break` is unrelated to using `break` to terminate a case in a `switch` block. The two situations are very different.

There might be times when you want to terminate not the entire loop, but just the current pass through the loop. You do this with the `continue` statement. The following example uses `continue` in a loop that prints out the square and cube of every number from 1 through 20, except 8:

```
int i;
for (i=1; i<=20; i++)
{
    if (i == 8)
        continue;
    int squared = i * i;
    int cubed = squared * i;
    System.out.println(squared + ", " + cubed);
}
```

The `continue` statement causes control to jump to the end of the loop body. Then the update (`i++`) is executed, the condition is checked, and perhaps more passes are made through the loop body. In other words, the current pass through the loop body is terminated prematurely. As with `break` statements, you can use `continue` statements with `do` and `do-while` loops as well as with `for` loops.

The `continue` statement allows you to improve on the preceding paycheck example, which broke out of the loop as soon as you couldn't afford to pay a salary. This was possibly unfair to the workers who had not yet been paid. After all, the employee who caused the break might have had the highest salary in the company. Even if there was not enough money to pay that person, there might still be enough left to pay someone else. So a more fair version of the program would be

```
int id;
for (id=1001; id<=1100; id++)
{
    float pay = getPayAmount(id);
    if (balance-pay < 0)
        continue;
    printCheck(id, pay);
    balance -= pay;
}
```

The only difference between this version and the previous one is the replacement of `break` with `continue`. Now the loop never terminates prematurely, although some passes through loop body might do so.

Nesting

The body of a loop can contain any valid Java code, including another loop, which can itself contain any valid Java code, including another loop, and so on. The technique of putting a loop within a loop is called *nesting*.

As an example of loop nesting, suppose you are writing code to generate frames for an animated movie. Assume that a frame consists of a grid of 1000 x 1000 pixels. (*Pixel* is an abbreviation for *picture element*. A pixel is a tiny dot of color, almost too small to see. If you hold a magnifying glass up to your computer screen, you can see the individual pixels.) Assume also that there is a method called `computePixel`, which takes as arguments the horizontal and vertical positions of the pixel whose color value is to be computed. Fortunately, `computePixel` also stores the color value in the appropriate place, so all you have to worry about here is calling the method with the right arguments.

The following code uses nested for loops to call `computePixel` for every pixel position:

```
1. int x, y; // x = horiz, y = vert
2. for (y=0; y<1000; y++)
3.     for (x=0; x<1000; x++)
4.         computePixel(x, y);
```

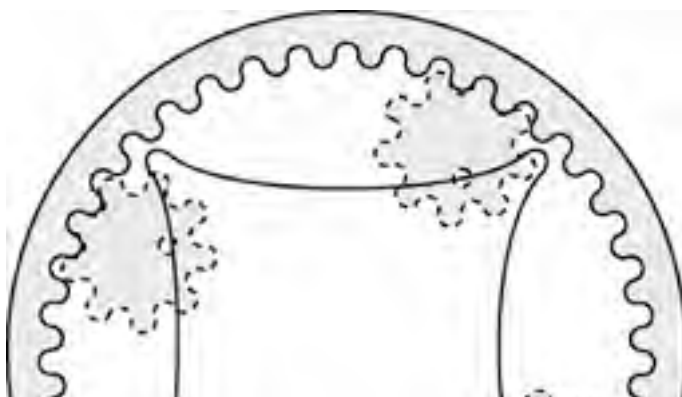
It is conventional to use `x` as a variable name for representing horizontal positions, and `y` for representing vertical positions. Line 2 says that the outer loop body will be executed with `y` ranging from 0 through 999. The outer loop body is lines 3 and 4. Line 3 says that the inner loop body, which is line 4, will be executed with `x` ranging from 0 through 999. So the first value pair passed to `computePixel` at line 4 will be (0, 0), followed by (1, 0), and then (2, 0), and then (3, 0), and so on up to (999, 0). Those thousand calls are the first pass through the outer loop. Then `y` is incremented from 0 to 1 and compared to 1,000. Since `y` is found to be still less than 1,000, the second pass through the outer loop begins: `computePixel` is called with arguments of (0, 1), (1, 1), through (999, 1). Every pass through the outer loop entails a thousand passes through the inner loop, until finally `computePixel` is called with arguments of (999, 999). At this point, the outer loop's condition is `false`, so the outer loop is finally done.

The body of the outer loop is two lines long (lines 3 and 4), but due to a technicality, the lines do not have to be enclosed in curly brackets. This is because, technically speaking, lines 3 and 4 are a single statement: a for loop. Only bodies consisting of multiple statements need to be enclosed in curly brackets. The precise definition of a statement is extremely intricate, but statements are easy to recognize because they end with semicolons. So you can get away with omitting curly brackets in this example, although you should indent responsibly to make it clear to readers that you are using a nested loop. However, it does no harm to add the curly brackets anyway (it's okay to have a block that contains just a single statement). The curly brackets do not affect execution speed, and they make the code a bit more readable, as you can see here:

```
1. int x, y; // x = horiz, y = vert
2. for (y=0; y<1000; y++)
3. {
4.     for (x=0; x<1000; x++)
5.     {
6.         computePixel(x, y);
7.     }
8. }
```

The `NestedLoopLab` animated illustration lets you use loops to draw cycloids. Cycloids are beautiful, complex geometric shapes. If you have ever played with a Spirograph, you have already appreciated cycloids. You could create some wonderful images with a Spirograph by drawing several curves in the same space but varying the curves' orientation or some other feature. If you have done this, you have performed a repetitive complicated task, varying features from one repetition to the next. In other words, you have done something that can be modeled with a loop, or possibly with nested loops.

A cycloid is the curve traced out by a point on a circle (the *roller*) as it rolls without slipping around the inside of a larger circle (the *gasket*). The roller always touches the gasket at one point, as shown in [Figure 5.5](#).



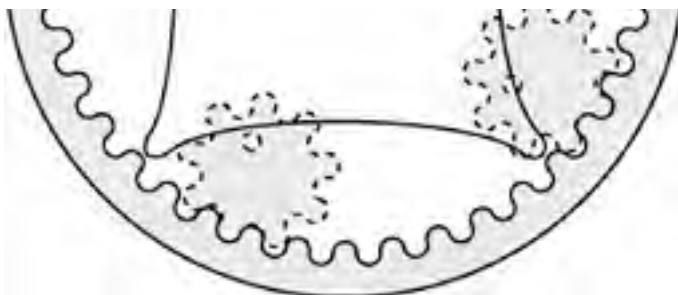


Figure 5.5: A cycloid

The ratio between the size of the roller and the size of the gasket determines the number of lobes the curve will have. The ratio in [Figure 5.5](#) is 1:4, so the curve has 4 lobes.

The *inset* is the distance from the tracing point to the rim of the roller. NestedLoopLab uses arbitrary inset units of 0 through 10, where 0 is the rim of the roller and 10 is the center. The orientation is the point of initial contact between the roller and the gasket, measured in clockwise degrees from straight up.

NestedLoopLab lets you select a ratio and color. You also choose a loop style. The default is no loop, but you can choose Loop to select a loop that varies the inset or the orientation. For really sophisticated images, you can use nested loops that vary the inset and the orientation, in either order. The Color choice lets you leave the color constant or vary the color in any of the loops

To launch NestedLoopLab, type `java loops.NestedLoopLab`. You will first see the display shown in [Figure 5.6](#).



Figure 5.6: NestedLoopLab: initial display

[Figure 5.7](#) shows NestedLoopLab with a ratio of 8:15 and a small inset.

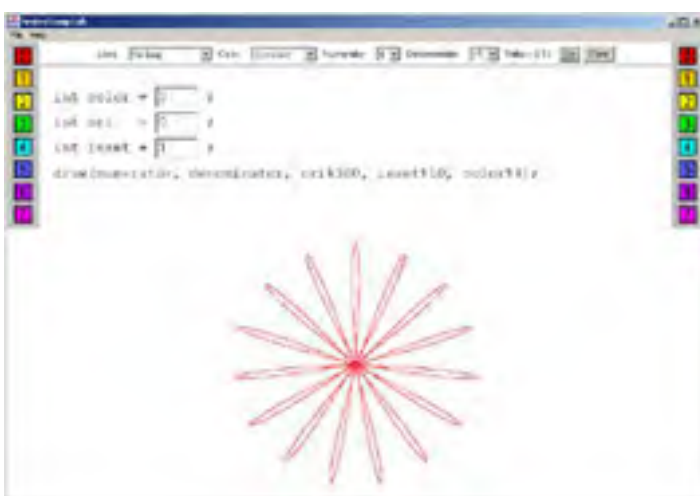


Figure 5.7: NestedLoopLab: 8:15

In [Figure 5.8](#), the configuration uses a loop, with the inset ranging from 0–6.

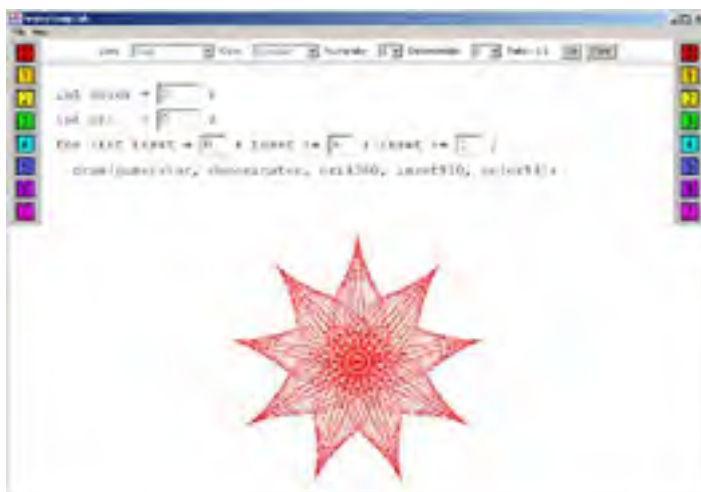


Figure 5.8: NestedLoopLab with a loop



Figure 5.9: NestedLoopLab with nested loops

Now try it for yourself. Enjoy! You can use the File , Gallery menu to view a few sample patterns. If you come up with a really spectacular image, please e-mail its settings to me at www.sybex.com so we can include it in future revisions of the gallery.

Labeled *break* and Labeled *continue*

Breaking out of a hierarchy of nested loops can be difficult. It might happen that code in an inner loop detects a condition that should cause an outer loop to be terminated. For example, you might use three nested loops to print paychecks for every employee in every department in every division of a large company. (Each department uses its own set of employee IDs, starting from zero.) The `getPayAmount` method now takes three arguments: division, department, and ID. Let's assume another feature for this method: The `if` statement will return a negative value if the corporate database that it consults is down. When this happens, paycheck processing should be terminated at once.

The following code would not be correct:

```
1. int  divn, dept, nDepartments, nEmployees, id;
2. float  pay;
3.
4. for (divn=0; divn <nDivisions; divn++)
5. {
6.     nDepartments = getDepartmentCount(divn);
7.     for (dept=0; dept <nDepartments; dept++)
8.     {
9.         nEmployees = getEmployeeCount(divn, dept);
10.        for (id=0; id<nEmployees; id++)
11.        {
12.            pay = getPayAmount(divn, dept, id);
13.            if (pay < 0)
14.                break;
15.            printCheck(divn, dept, id, pay);
16.            balance -= pay;
17.        }
18.    }
19. }
```

The code assumes the presence of methods that return the number of departments in a division (`getDepartmentCount`) and the number of employees in a department (`getEmployeeCount`). It also assumes that `nDivisions` has been preset to the

number of divisions in the company. The variable names `nDivisions`, `nDepartments`, and `nEmployees` mean, of course, the number of divisions, the number of departments, and the number of employees. This kind of naming convention is common and useful.

Unfortunately, the code doesn't work. The `break` keyword breaks out of the immediately enclosing loop, not out of all loops. So the `break` at line 14 just breaks out of the innermost loop (lines 10-17). Processing then continues with the next department because we are still in the middle loop (lines 7-18).

There is a simple but clumsy solution, which is shown in the following code. The innermost loop, when it detects a database problem, sets a boolean variable to `true`. The middle and outer loops have to check this variable and do their own break if it is `true`. Incidentally, a boolean variable that's used in this way to indicate program status is often called a *flag*. Here is the correct but clumsy code:

```
1. int    divn, dept, nDepartments, nEmployees, id;
2. float  pay;
3. boolean trouble = false;
4.
5. for (divn=0; divn <nDivisions; divn ++)
6. {
7.     nDepartments = getDepartmentCount(divn);
8.     for (dept=0; dept <nDepartments; dept ++)
9.     {
10.        nEmployees = getEmployeeCount(divn, dept);
11.        for (id=0; id<nEmployees; id++)
12.        {
13.            pay = getPayAmount(divn, dept, id);
14.            if (pay < 0)
15.            {
16.                trouble = true;
17.                break;
18.            }
19.            printCheck(divn, dept, id, pay);
20.            balance -= pay;
21.        } // End of inner loop
22.        if (trouble)
23.            break;
24.    } // End of middle loop
25.    if (trouble)
26.        break;
27. } // End of outer loop
```

The code is difficult to read, which is a good indicator of code that is needlessly complicated. The solution is Java's *labeled break* feature. This feature lets you assign a name, or *label*, to any `for`, `while`, or `do-while` loop. The label is any valid identifier (so you just need to follow the same rules that govern variable or method names). The label, followed by a colon, appears just before the `for`, `while`, or `do` keyword. Now you can break out of the labeled loop, even from code in a loop nested inside the labeled loop, by adding the loop's label after the `break` keyword.

The following code elegantly fixes the paycheck program by using a labeled loop and a labeled break:

```
1. int    divn, dept, nDepartments, nEmployees, id;
2. float  pay;
3.
4. bigloop: for (divn=0; divn <nDivisions; divn ++)
5. {
6.     nDepartments = getDepartmentCount(divn);
7.     for (dept=0; dept <nDepartments; dept ++)
8.     {
9.        nEmployees = getEmployeeCount(divn, dept);
10.        for (id=0; id<nEmployees; id++)
11.        {
12.            pay = getPayAmount(divn, dept, id);
13.            if (pay < 0)
14.                break bigloop;
15.            printCheck(divn, dept, id, pay);
16.            balance -= pay;
17.        }
18.    }
19. }
```

Now line 14 causes the outermost loop to break.

Java also provides a labeled `continue` feature. The statement `continue label;` terminates the current pass through the labeled loop. If the label were omitted, the current pass through the innermost loop would terminate instead.

Loops and Scope

The [previous chapter](#) introduced the concept of scope. As a reminder, a variable's scope is the block (inside curly brackets) that most tightly encloses the variable's declaration. *The variable has definition only within its scope.* Outside the scope, the variable's name may be reused, but the name refers to a different piece of data with its own scope.

With the introduction of loops, you begin to use code that can consist of blocks nested in blocks nested in blocks, and so on. This raises the issue of where variables should be declared. A good rule of thumb is that a variable's scope should be as small as possible. This means that if a variable is used only in a loop, it should be declared inside the loop.

For example, in the paycheck code of the [previous section](#), you declared `float pay` outside the outermost loop, even though it is only used in the innermost loop. A clear approach would be to eliminate the declaration on line 2 and change the innermost loop to the following:

```
for (id=0; id<nEmployees; id++)
{
    float pay = getPayAmount(divn, dept, id);
    if (pay < 0)
        break bigloop;
    printCheck(divn, dept, id, pay);
    balance -= pay;
}
```

Now anyone who reads the code and wonders where `pay` is used only has to think about five lines.

You are allowed to declare a variable in the initialization statement of a for loop. Thus, the following is allowed (and, in fact, is encouraged):

```
for (int i=0; i<10; i++)
{
    // Loop body
    // More loop body
}
```

The scope of `i` is the body of the loop.

Team LIB

PREVIOUS NEXT

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Rewrite the following code to maximize readability:

```
switch (x)
{
    case 100:
        System.out.println("x is big");
        break;
    case 101:
        System.out.println("x is big");
        break;
    case 10:
        System.out.println("x is medium");
        break;
    case -1000:
        System.out.println("x is negative");
        break;
}
```

2. Rewrite the following code to make it cleaner:

```
boolean flag = false;
switch (a)
{
    case 1:
        x = 1000;
        flag = true;
        break;
    case 30:
        y = 1000;
        flag = true;
        break;
}
if (!flag)
    z = 1000;
```

3. What happens when the following code is executed with `val` equal to 10? 100? 1,000? First, decide just by looking at the source code. Then write a program to verify your answer.

```
switch (val)
{
    case 10:
        System.out.println("ten");
    case 100:
        System.out.println("hundred");
    default:
        System.out.println("thousand");
}
```

4. Run the WhileLab animated illustration by typing `java loops.WhileLab`. Try changing the value in the condition in the third line. What do you notice about the final value of `a`?
5. The description of WhileLab suggests three exercises, which are repeated here. For each desired result, configure the inputs of WhileLab to produce that result. Then verify your work (and make sure WhileLab is trustworthy) by writing an application that duplicates each while loop. The loops should generate the following results:
 - The sum of the numbers 1 through 500, inclusive.
 - The sum of the even numbers from 50 through 60, inclusive.
 - The product of the first 5 odd numbers.
6. There is a number game called Hotpo that can entertain you for a few minutes while you're stuck in traffic, waiting for a movie to start, or having dinner with someone really boring. Hotpo stands for Half Or Triple Plus One, and it works like this: Think of an odd number. Now mentally calculate another number, as follows: If the first number was even, the next number is half the first one; if the first number was odd, the next number is 3 times the first number, plus 1. Now you can forget the first number and apply the Half Or Triple Plus One formula to your current number. Keep going until the value reaches 1. Let's try this with a starting number of 5. The series is $5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$.

Write a program that plays Hotpo. First, initialize a variable called `n` to the starting value you're interested in. Then enter a loop that prints out each number in the sequence, along with the current step number. For example, the output for 3 would be

```
Step #1: 10
Step #2: 5
Step #3: 16
Step #4: 8
Step #5: 4
Step #6: 2
Step #7: 1
```

Should the program use a while loop or a for loop?

7. What is the value of n after the following code is executed?

```
int n = 1;
outer: for (int i=2; i<10; i++)
{
    for (int j=1; j<i; j++)
    {
        n *= j;
        if (i*j == 10)
            break outer;
    }
}
```

Team LIB

PREVIOUS NEXT

Chapter 6: **Arrays**

Overview

All of the data types presented so far in this book have represented single units of information. The `char` type represents a single character, while the other types represent numbers with various formats and ranges.

This chapter will introduce arrays, which are clusters of data. You will learn how to create arrays and how to use them in programs, especially in the context of loops.

Arrays are extremely basic examples of objects. We can't claim that when you master arrays, you will have mastered object-oriented programming. However, in the course of this chapter, you will learn a number of concepts that will make it easy for you to master objects when they are presented in the next several chapters.

Clusters of Data

An *array* is a cluster of variables, called *components*, all of the same type. The array has a name, but the individual variables within the array do not. Each of the components has a unique identifying integer, called an *index*. The plural of index is *indices*, which proves that someone was paying attention in Latin class. The indices range from 0 through $n-1$, where n is the number of components in the array.

Before you use an array, you have to declare its type and create it. You have already seen type declarations in the context of primitive data types, and array declaration is quite similar. Creation is a new concept, and we will discuss it in some depth.

Declaring Arrays

Array declaration, like primitive declaration, associates a variable name with a data type. There are two ways to declare an array. The preferable format is

```
Component_type[] name;
```

Note the square brackets after the component type. An example of this format is

```
float[] temperaturesCelsius;
```

This declaration says that `temperaturesCelsius` is the name of an array whose components are all of type `float`. The number of components will be specified later, when the array is created. Note the plural in the name, indicating that the variable relates to more than one temperature.

The alternative format for array declaration is

```
Component_type name[];
```

The only difference is that the empty square brackets now come after the name, rather than after the component type. This format is included as a holdover from older programming languages (C and C++). It is considered less readable than the first format, and we will not use it in this book.

Creating Arrays

At some point after you declare an array, you need to create it. This is new. With primitives, all you had to do was declare a variable's type, and the variable came into existence. Arrays, as well as all other kinds of objects, are different: You have to create them explicitly. This is done with the keyword `new`.

The format of an array creation statement is

```
name = new component_type[number_of_components];
```

In the [previous section](#), you declared a variable named `temperaturesCelsius` to be an array whose components have `float` type. From this point on, we will say this more briefly: `temperaturesCelsius` is an array of floats. When you want to create the array, you first have to decide how many components it will have. If you want 10 components, for example, you would create the array like this:

```
temperaturesCelsius = new float[10];
```

The array size does not have to be a literal int. A variable is acceptable. Suppose you have a method called `getNTempReadings`, which returns the number of temperatures available to the program. Then you might do the following:

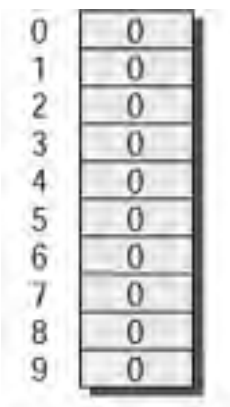
```
int nTemps = getNTempReadings();  
temperaturesCelsius = new float[nTemps];
```

You could even get more terse:

```
temperaturesCelsius = new float[getNTempReadings()];
```

When an array of primitives is created, all of its components are given an initial value. Numeric arrays (that is, arrays of `byte`, `short`, `int`, `long`, `float`, and `double`) have all components initialized to 0. Boolean arrays have all components initialized to `false`. Char arrays have all components initialized to the null character, which is a non-printing, do-nothing zero value that indicates "no character at all."

It is convenient to represent an array as shown in [Figure 6.1](#).



temperaturesCelsius

Figure 6.1: A new array

Figure 6.1 shows the array after it has been created but before any of its components have been modified. Now is the perfect time to learn how to modify the components.

Using Array Components

The components of an array of n components have indices from 0 through $n-1$. The names of those components are `arrayName[0]`, `arrayName[1]`, and so on, through `arrayName[n-1]`. Thus, in this example, you could use the following code to set the first and last components to -10 and 10, respectively:

```
temperaturesCelsius[0] = -10;  
temperaturesCelsius[9] = 10;
```

Sometimes the first component is called the *zeroth* component, so that the adjective will match the index. The term eliminates confusion, because one could reasonably (but wrongly) believe that the first component is `temperaturesCelsius[1]`.

Figure 6.2 shows the array after the preceding code has been executed.

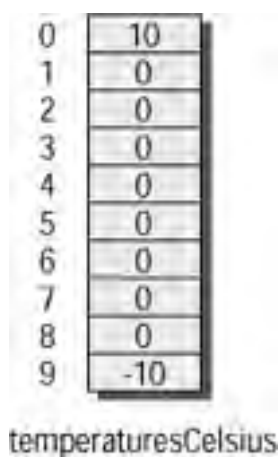


Figure 6.2: A used array

Of course, you can also read the value of an array component. The following code sets a component to the average of two other components:

```
temperaturesCelsius[5] =  
    (temperaturesCelsius[6] + temperaturesCelsius[7]) / 2;
```

You can see that an individual array component can be used in any context where a primitive variable of the same type can be used.

Array Length

When you create an array, you specify the number of components. Subsequently, the array knows its component count. You can read the number of components with the expression `arrayName.length`. For example, if you have an array called `employeeNames`, the following code sets an `int` called `nEmployees` to be the length of the array:

```
nEmployees = employeeNames.length;
```

The array's length is permanently fixed at creation time, so you can't modify it. The following code would cause a compilation error:

```
employeeNames.length = 5000;
```

Array Initialization

We have already said that when an array is created, its components are initialized to zero for numeric types, or to `false` or the null character for booleans and chars. If you want the array to start life with different contents, you can set the values of the components you want to change one by one, such as follows:

```
char[] chars = new char[10];  
chars[3] = 'L';  
chars[4] = 'C';
```

If you want to specify a value for all the components of the new array, a more compact syntax is available:

```
name = new type[] {value0, value1, ... };
```

The compiler determines the array length by counting the values in the curly brackets. The array components are initialized to those values. For example, the following code creates and initializes an array of 4 bytes:

```
byte[] bytes = new byte[] { 3, 5, 7, 99};
```


Arrays and Loops

Loops, and especially *for* loops, are ideal for processing arrays. No matter what you want to do to the array, you typically use a *for* loop with a loop counter that ranges from 0 through `array-length-minus-1`. Within the loop's body you perform whatever processing you want, using the loop counter as an array index.

For example, the following code computes the product of all the values in an array called `measurements`:

```
double product = 1;
for (int i=0; i<measurements.length; i++)
    product *= measurements[i];
```

Note that this code works on arrays of all sizes, because it reads the array size from `measurements.length`.

For another example, let's revisit the paycheck-printing code from the [previous chapter](#). Here is one of the several versions of that code:

```
int id;
for (id=1001; id<=1100; id++)
{
    float pay = getPayAmount(id);
    printCheck(id, pay);
    balance -= pay;
}
```

The code is a bit unrealistic, because in a company with 100 employees, people are going to join or leave the company. The number of employees and their individual IDs are going to change. However, it is reasonable to assume that there could be a method called `getIDsFromDatabase`, which queries the corporate database and returns an array of `int` containing the ID of every employee who should get a paycheck. Then the preceding code would be modified as follows:

```
int[] ids = getIDsFromDatabase();
for (int id=0; id<ids.length; id++)
{
    float pay = getPayAmount(id);
    printCheck(id, pay);
    balance -= pay;
}
```

Multi-Dimensional Arrays

The arrays we have looked at so far have been *one-dimensional*. This means that each component is specified by a single unique index. Java also supports *multi-dimensional* arrays, in which each component is specified by a unique sequence of indices. The number of indices in the sequence is the array's *dimension*. In a two-dimensional array, for example, each component has two indices. [Figures 6.1](#) and [6.2](#) portrayed some one-dimensional arrays as columns of boxes. We can portray a two-dimensional array as a lattice or matrix, where each component is identified by its row and column, as in [Figure 6.3](#).

45	0
9	39
16	93

Figure 6.3: A two-dimensional array

If the array in [Figure 6.3](#) were called `twoDimInts`, it would be declared as

```
int[][] twoDimInts;
```

The array is now declared but has not been created yet. When you create an n-dimensional array, you have to specify n sizes: one size for each dimension. To create a two-dimensional array of 3 rows and 2 columns, as in [Figure 6.3](#), use the following code:

```
twoDimInts = new int[3][2];
```

Now the array has been created. At creation time, every component is initialized, as with one-dimensional arrays. To access an individual component, you use the array name followed by both indices, with each index in square brackets. For example, the following code initializes every component of `twoDimInts` to 39:

```
for (int i=0; i<3; i++)
  for (int j=0; j<2; j++)
    twoDimInts[i][j] = 39;
```

Suppose you have 50 weather stations, each of which takes a temperature reading every hour throughout one day. You might store the data in a two-dimensional float array called `temps`, where `temps[t][s]` is the temperature at time `t` recorded by station `s`.

The following code could be used to print the average temperature over all stations, hour by hour:

```
for (int hour=0; hour<24; hour++)
{
  float tempTotal = 0;
  for (int stn=0; stn<50; stn++)
    tempTotal += temps[hour][stn];
  float tempAvg = tempTotal / 50;
  System.out.println("Average temp at time " + hour +
    " = " + tempAvg);
}
```

On the other hand, you might want the average temperature over the entire day for each station. For that, you would use the following code:

```
for (int stn=0; stn<50; stn++)
{
  float tempTotal = 0;
  for (int hour=0; hour <24; hour++)
    tempTotal += temps[hour][stn];
  float tempAvg = tempTotal / 24;
  System.out.println("Average temp at station " + stn +
    " = " + tempAvg);
}
```

These examples show that processing a two-dimensional array generally requires a two-deep nested loop. In general, processing an N-dimensional array requires an N-deep nested loop.

The `BoolArrayLab` animated illustration uses a two-deep nested loop to let you set the values in a 200-by-200 boolean array. The array contents are illustrated by a grid of 200 by 200 pixels. A blue pixel represents a value of `true`; a black pixel represents `false`. (You probably can't see the individual pixels unless you use a magnifying glass.) To run the program, type `java arrays.BoolArrayLab`. The code looks like this:

```
boolean[][] bools = new boolean[200][200];
for (int y=0; y<200; y++)
  for (int x=0; x<200; x++)
    bools[x][y] = _____ ;
```

You supply the formula in the last line. The formula can be any valid boolean expression. Initially, the program comes up with the following formula:

```
bools[x][y] = x>y;
```

[Figure 6.4](#) shows `BoolArrayLab`'s initial screen.



Figure 6.4: BoolArrayLab

The File , Gallery menu offers 7 sample formulas, and you are encouraged to try your own or modify the ones provided. [Figure 6.5](#) shows a parabola.

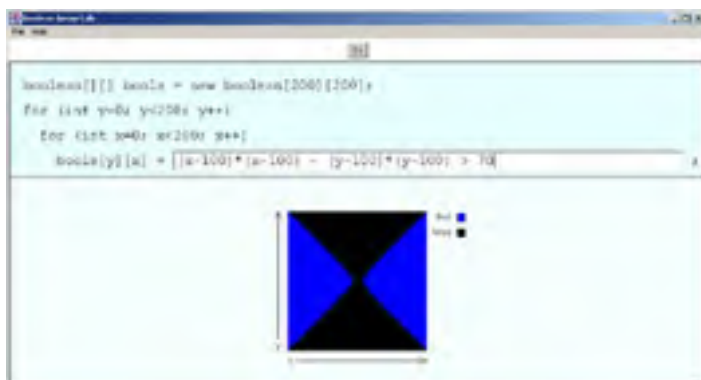


Figure 6.5: BoolArrayLab drawing a parabola

The lower portion of the display renders the contents of the array. Since 200 x 200 is fairly large, the display uses a rectangular grid of 200 x 200 pixels. An array component with a value of `true` is represented by a blue pixel, while a `false` value is represented by a black pixel.

Before you launch the program, can you guess what sort of image is produced by the initial formula `bools[x][y] = x>y;`? If you discover a formula that produces an interesting display, please email it to www.sybex.com. I will use it and mention your name in the next edition.

Arrays as Objects

Now you know enough about arrays to write some very useful code. At this point, we will stop discussing the syntax and use of array code. It's time to look at what you might think of as *array anatomy*. This material is extremely important, because nearly everything you'll learn about array anatomy also applies to the anatomy of full-blown objects. If you understand the material in the remainder of this chapter, you will have a good solid foundation for learning about object-oriented programming in Java.

You have already seen that declaring an array is different from declaring a primitive. When you declare a primitive, the variable is right there for you. But when you declare an array, you still have to create it. This is a clue that there is more going on with arrays than with primitives.

You can think of memory as being divided into two parts: *accessible* and *inaccessible*. (This is unofficial terminology, but it is very useful and will be used throughout this book.) Primitives exist in accessible memory; arrays exist in inaccessible memory. [Figure 6.6](#) shows memory divided into its two parts, populated with a few primitives and arrays.

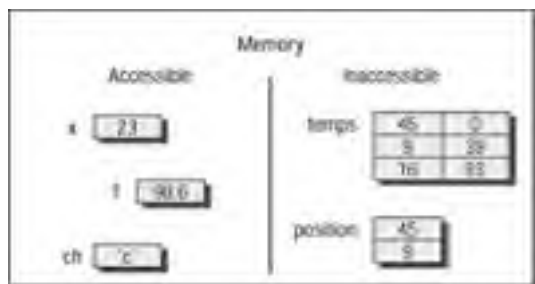


Figure 6.6: Accessible and inaccessible memory

Any variable on the left side of the figure – that is, in accessible memory – can have its values read and written. Accessible memory is for primitive variables, and for a kind of variable that you have already used without knowing it. This kind of variable is called a *reference*. References are used for all access to arrays. When you declare an array, what gets created is just a reference. (Remember, the array is not created until you say `new`.) The reference exists in accessible memory. No matter what kind of array you declare – no matter what type, size, or number of dimensions it has – the reference is 32 bits wide.

When you create an array by invoking the keyword `new`, space for the array is reserved (or *allocated*) in inaccessible memory. For example, the code `int[][] ages = new int[3][2];` would cause allocation of 24 bytes (3 times 2 ints, times 4 bytes per int), as shown in [Figure 6.7](#).

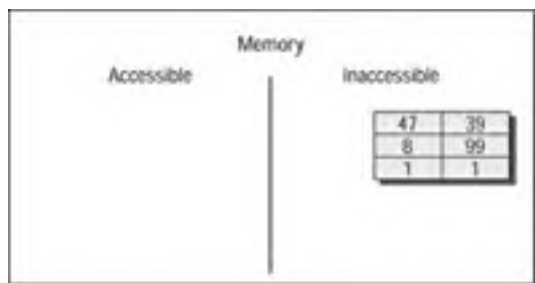


Figure 6.7: An array of bytes in inaccessible memory

Invoking `new` is a bit like invoking a method: The code returns a value. The value returned by `new` is almost – but not quite – the address in inaccessible memory of the freshly created array. If you are at all unclear about the distinction between memory address and memory contents, you might want to return to [Chapter 1](#) and play with the SimCom animated illustration.

Actually, it doesn't matter if the value returned by `new` is exactly the address of the array, or almost-but-not-quite the address, or only vaguely related to the address. The details of the relationship are a hidden part of the Java Virtual Machine, and they may even vary from one implementation of the JVM to another. For this reason, the value is called a *reference* to the array. *Reference* implies that the value uniquely identifies the array in a way that is hidden from us.

Now we can look at what really happens when the following code is executed:

```
int[][] ages; // Allocation
ages = new int[3][2]; // Construction & ref assignment
```

The allocation line creates a reference named `ages` in accessible memory. Then the creation line causes space for the array to be allocated in inaccessible memory. The invocation of `new` returns a reference to the array; this reference is the right-hand side of the `=` assignment. The reference is then stored in the variable whose name appears on the left side of the `=` assignment, namely `ages`. This situation is illustrated in [Figure 6.8](#).

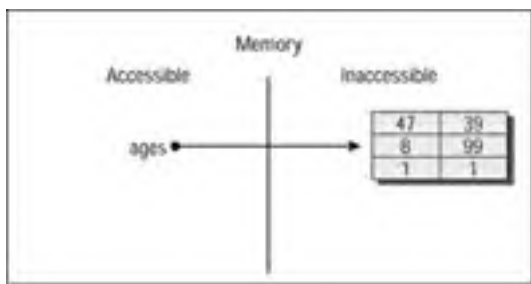


Figure 6.8: Reference and array

Notice that the comment on the second line of code above says *Construction* rather than *Creation*. *Construction* is the technical name for creating something in inaccessible memory by invoking `new`.

Why does this matter? So far, the array code we have presented has made sense without burdening you with all this reference stuff. But there are certain very useful operations you can do with arrays that only make sense if you understand references. These are operations that you have already seen in the context of primitives: assignment, and argument passing.

Suppose `ages` and `otherAges` are declared to be arrays of the same type. What does it mean to say the following?

```
ages = otherAges;
```

Contrary to reasonable expectation, this code emphatically does *not* create a new array whose components have the same values as the original array. Remember that `ages` and `otherAges` are really references. So `ages = otherAges;` just copies the 32-bit pattern from one reference to the other. The result is a second reference that (in some sense) points to the same thing the first reference pointed to: the array. You now have two references to the same array, as shown in Figure 6.9.

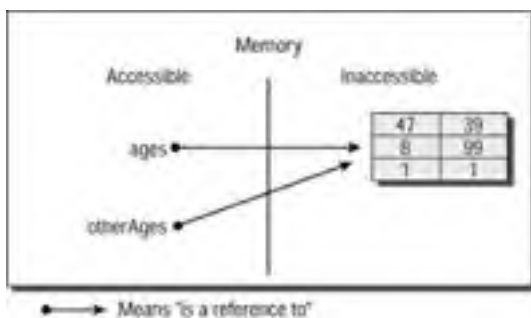


Figure 6.9: Two references, one array

The `CreateArrayLab` animated illustration dynamically illustrates the following code:

```
double d = 1.23;
double e = d;
d = 3003;
double[] doubleArray = new double[4];
double[] theCopy = doubleArray;
doubleArray[1] = 98.6;
e = theCopy[1];
```

Start the program by typing `java arrays.CreateArrayLab`. You will see the display shown in Figure 6.10.

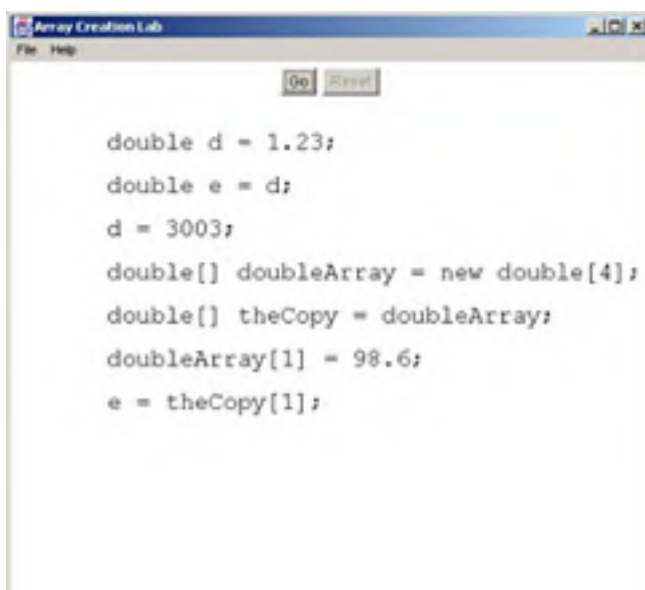




Figure 6.10: CreateArrayLab

Click on the Go button to see the animation. When the animation finishes and you want to watch it again, you can click on Reset to return the display to its original state.

References usually point to arrays (or to objects, as we will see later). However, there is a special value that can be assigned to any reference. The value is `null`, and it indicates that the reference does not point to anything. For example, you might make the following declaration and assignment:

```
double[] doubles;  
doubles = null;
```

The `null` reference value is only slightly useful at the moment, but you will see a use for it in the section on garbage collection later in this chapter. You will also make extensive use of `null` in later chapters, in the context of objects.

Passing References to Methods

Now you understand what really happens in code like the following:

```
float[] floatArray = new float[4];  
float[] theCopy = floatArray;
```

Are you likely to encounter this situation? Are you likely to make a copy of a reference, when you already have a perfectly good one, given the confusion that a copy might create? Actually, yes. Within a single method, there isn't really any good reason for copying a reference. However, you might want to pass an array as a method argument.

Remember that when you pass a primitive as an argument to a method, the method actually gets a copy of the primitive. Thus, the method can modify the copy, and the caller will never be aware of the modification because the caller has no access to the modified copy.

With arrays and methods, the situation is a bit different. You don't actually pass an array into a method. You pass a reference to the array. The method receives a copy of the reference. The caller's original reference and the method's copy are identical 32-bit patterns, so they both (in some sense) point to the same object in inaccessible memory: the array. So when you pass an array reference as a method argument, the method can use the reference to modify the array, and the modifications *will* be visible to the caller.

The PassArrayLab animated illustration animates the following code:

```
int[] intarray = new int[3];  
setInts(theArray);  
.  
.  
static void setInts(int[] ints)  
{  
    for (int n=0; n<ints.length; n++)  
        ints[n] = 22;  
    return;  
}
```

The `return` statement isn't really required, since the method's type is `void` and it would return anyway after executing its last line. The `return` is just there to make the animation more clear. When a method returns, all variables that were declared within the scope of the method cease to exist. This includes the method's arguments (`ints` in this example). The space in accessible memory that was allocated for the variables is reclaimed by the system. Conceivably, the next variable to be declared could occupy the same bytes that used to constitute the `ints` argument. But all is not lost. Although the `ints` argument ceases to exist, the array it references continues to exist.

Invoke PassArrayLab by typing `java arrays.PassArrayLab`. The Go and Reset buttons start the animation and reset the display. Run the animation a few times, until you are confident that you understand that what gets passed to the method is a reference and not an array. That way, changes made to the array are permanent and visible to the caller.

Garbage Collection

Garbage collection is a mundane term for a very important feature.

You have seen that arrays are created by invoking the keyword `new`. Surely there must be a way to recycle an array's memory after the array is no longer needed. Something like this happens to the arguments and local data in a method, when the method returns. But in that case, the recycled data consists of primitives and references. In other words, it's data in accessible memory that was reserved by declaring arguments or variables. In the case of arrays, we are concerned with data in inaccessible memory that was reserved by invoking `new`.

Java's precursor languages, and in particular C and C++, required the programmer to explicitly free up memory that was no longer needed. This was the only way that the memory could become available for reuse. This led to problems. One such problem is called a *memory leak* bug. If a bug causes a method to neglect to free up a few hundred bytes, that's not much of a problem. The

program is probably running on a system with at least several million bytes of available memory, so if a few hundred become unavailable, there is still plenty left. But if the method is called in a loop that executes 1,000 times, a few hundred thousand bytes become unavailable. That might have an impact. If the loop executes 1,000 times every hour, eventually there will be no more memory available for allocating new arrays, and the program will crash. The problem is called a memory leak because the pool of available memory gradually diminishes.

Java makes memory leaks highly unlikely, because in Java the programmer never decides when to recycle unneeded arrays and objects. The JVM decides when memory is no longer needed, and such memory is automatically recycled. The JVM uses the following logic to decide when to recycle memory:

When an array (and, as we'll see later, an object) is created, inaccessible memory is created and a reference is returned. The JVM keeps track of how many references are pointing to an array or other object. Your program might make copies of a reference, and might pass the reference as a method argument. As long as there is at least one reference to something, there is a chance that you might want to use that particular something, so its memory will not be recycled. However, when the last reference to an object ceases to exist, suddenly there is no way to read or write the object, or to use it in any way. You can't talk about something if you have no name for it. Since the unreferenced object can no longer play any role in your program, its memory will be automatically recycled. Such automatic recycling of unneeded memory is called *garbage collection*.

Consider the following method:

```
1. void useAnArray(int size)
2. {
3.     int[] theArray = new int[size];
4.     int[] aRefCopy = theArray;
5.     int[] anotherCopy = theArray;
6. }
```

Line 3 constructs an array of ints and stores the returned reference in `theArray`. Line 4 copies the reference, so after line 4 there are 2 references to the array. After line 5, there are 3 references to the array, but only briefly. Immediately after line 5, the method returns, so all its variables are recycled. Suddenly, instead of 3 references to the array, there are none at all. Now the program no longer has any way to access the array, which will be recycled shortly.

If you have an array that you no longer need, there is no explicit way to recycle its memory. However, you can usually set all references to the array to `null`, which will cause garbage collection.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. The following two declarations are equivalent as far as the compiler is concerned, but one is considered more readable than the other. Which is more readable, and why?
 1. `double dubs[];`
 2. `double[] dubs;`
2. Write a line of code that declares an array of 5 ints and initializes the array to contain the first 5 prime numbers. The code should be a single statement.
3. Write a method whose single argument is an array of double. The method should return the average (mean) of the array's components. Write an application that tests the method by passing it an array containing any values you like.
4. Write a program that uses the array-averaging method of Question 3. The program should compute and print out the average of an array (you can choose the component values). Then the program should add 100 to each component, and again compute and print out the average.
5. Write a program that contains a method that creates and returns an array of int containing the first n square numbers, where n is the method's argument. Test your method by calling it with $n=10$. Your program should print out the index and value of each component, in *descending* order.
6. Write a method that creates a multiplication table. The method should return a two-dimensional array of N by N ints, where N is specified by the method's argument. In the array, the component at `[row][col]` should have a value of `row*col`.

Chapter 7: Introduction to Objects

The [previous chapter](#) presented arrays, which are Java's simplest kinds of objects. Arrays are much less sophisticated than other kinds of objects. Usually when people say "object," they mean it in a casual sense that excludes arrays. A program whose only objects are arrays can hardly be called object-oriented. However, in learning about arrays, you have learned a number of concepts that are vital to your mastery of full-fledged objects. You are now ready to enter the world of object-oriented programming, perhaps never to return.

The animated illustrations for this chapter provide visual reinforcement for the concepts that will be presented here. Please be sure to run them and take the time to play with them when the text invites you to do so.

Arrays Versus Objects

Before we begin, let's agree on some terminology. In the most formal sense, an array is a kind of object. However, we are about to compare arrays and other objects, and we need to avoid cumbersome language. It would be useful to say, for example, "objects have data and methods, rather than, "objects that aren't arrays have data and methods." So for the remainder of this book, unless it will cause confusion, "object" will mean "object but not array."

You already know a lot about objects from your study of arrays. Here are some similarities between arrays and objects:

- Objects contain clusters of data.
- Objects are created by invoking the keyword `new`.
- Objects inhabit inaccessible memory.
- Objects are manipulated indirectly, via references.
- Object references can be passed as method arguments; objects cannot.
- Objects are not explicitly destroyed; they are garbage-collected when they have no more references.

Another recognizable feature of objects is the use of the period as a symbol to denote "property of" when it follows the name of an array or object. With arrays, the syntax `arrayReference.length` gave you the number of components in the array. With objects, the syntax `objectReference.something` gives you access to the extensive power and features of an object. You will see how this works in great detail later in this chapter.

Objects have many features that go far beyond what arrays can do. Here are some unique features of objects:

- They can contain data of different types.
- They can contain methods as well as data.
- They are related to classes.

Classes are among the most important concepts in object-oriented programming. They are actually quite simple to understand, as you will see in the [next section](#).

Classes

Plato would have approved of the concept of classes.

The easiest way to learn about classes is to step outside the domain of object-oriented programming for a moment and look at the real world. (Plato might not have approved of calling it "real.") We experience things in the world, and we create categories in our minds so that we can think about those things collectively. [Figure 7.1](#) illustrates this mental process.

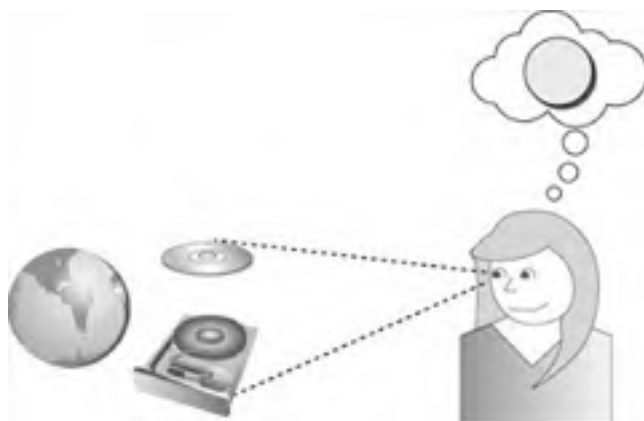


Figure 7.1: Class as mental category

As another example, "dog" is a category. In daily life, when dealing with the external world, we don't really experience the category "dog." We experience individual dogs, such as Harley or Sumo or Rover. So "dog" is a class or category, and the individual dogs Harley and Sumo and Rover are individual instances of that class.

Now back to object-oriented programming. In Java, a *class* is a piece of code that describes a category of thing that you want to represent in software. In fact, a Java program is just a bunch of interacting class definitions. You might have suspected as much, since every complete application listing we have seen so far has contained the mysterious keyword `class`. On the other hand, you might *not* have suspected as much. So far we have put a lot of effort into concealing the object-oriented nature of class code, because it was not yet time to talk about objects and classes. Well, now the time has come.

Suppose you want to write a program to model the behavior of Harley, Sumo, and Rover. First you spend some time thinking about what these three have in common. Eventually you realize that they are all dogs, so you decide to create a class called Dog. (In Java, a class name can be any valid identifier, but by convention we capitalize the first letter.) You create the class by writing a source file that looks like this:

```
public class Dog
{
    . . .
}
```

This is called a class definition. The code that goes between the curly brackets is the *body* of the class definition. Bodies can be as short as a few lines of code, for very simple classes. There is no upper limit on the size of the body, but typical large class bodies can be hundreds or even thousands of lines long. Of course, you don't yet know how to write a class body, but that is what the rest of this book is all about.

A class definition should appear in a file whose name matches the class name. So the Dog class should appear in a file called `Dog.java`. Compiling this file will result in a file called `Dog.class`. Now that you know what a class is, the `.class` filename extension makes sense. This is not an absolute rule, but explaining when you do and don't have to apply it would require presenting a number of concepts that are out of place here. If you are curious, please wait until [Chapter 9, "Packages and Access."](#)

So a class is something that you define when you write your source code. What about objects? An *object* is an individual instance of a class. Objects are created when your program is executed. More specifically, an object, like an array, is created by an invocation of the keyword `new`. The syntax for object creation is a bit different from the syntax for array creation, as you will see in the [next section](#).

Objects and Their Data

You have already learned that objects, like arrays, contain clusters of data. You have also learned that the data in an object, unlike the data in an array, can be of differing types. The number, types, and names of an object's data elements are all defined in the object's class definition.

Let's look at a very simple example. Here is a very simple class definition:

```
public class Person
{
    int    age;
    short weight;
}
```

This is our first example of a class that does not contain a method called `main`. In fact, this class definition has no methods at all. The class just defines a bundle of data. The bundle contains an `int` called `age` and a `short` called `weight`.

To create an individual instance of this class, you would use the following code:

```
Person keara;
keara = new Person();
```

The first line is a declaration. Like all other declarations, it tells the compiler that you will be using a variable called `keara` and it will be of a certain type. At first glance, it appears that the type of `keara` will be `Person`; this is almost true, but not quite. Actually, the declaration says `keara` will be a *reference* to an object, and that object will be an instance of the `Person` class. If the distinction seems subtle, it is also very important.

The situation is similar to what we saw in the [previous chapter](#), in the context of arrays. The declaration `int[] temperatures;` says that `temperatures` will be a reference (in accessible memory) to an array (in inaccessible memory). Similarly, the declaration `Person keara;` says that `keara` will be a reference (in accessible memory) to an array (in inaccessible memory). In both cases, the declaration does not cause construction of the array or object. Nothing gets constructed until `new` is executed.

When the second line (`keara = new Person();`) executes, an object is constructed, using the class definition as a kind of stencil or cookie cutter. The JVM knows which class definition to use (remember, a Java application can consist of many class files) because the class name appears after `new` and before the empty parentheses. As with arrays, the invocation of `new` constructs an object in inaccessible memory and returns a reference to that object. The reference is stored in the variable `keara`, so `keara` now refers to the newly created object. At this point, the situation is as shown in [Figure 7.2](#).

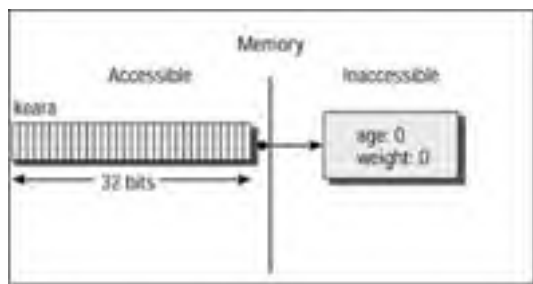


Figure 7.2: Reference and object

The figure shows that the object contains its own set of data variables, with names and types as specified in the class definition source file. When an object is constructed, its data variables (called *fields*) are initialized in the same way array components are initialized. Numeric fields are initialized to 0, char fields are initialized to the null character, and boolean fields are initialized to false.

Now you can use the reference `keara` to manipulate the object's fields. The following code writes and reads the fields of an object, using a new notation:

```
keara.age = 8;
short f = keara.weight;
```

In both lines, you use the following syntax to refer to an object's field:

```
Object_reference.field_name
```

The period is pronounced "dot." So if you were reading the first code line out loud, you would say, "Keara dot age equals eight."

The DataLab animated illustration shows the construction of an object and the use of a reference to access fields of that object. Please take a moment now to run the animation by typing `java objects.DataLab`. [Figure 7.3](#) shows DataLab's initial display.



```
public class Person
{
    int age;
    float weight;
}

// ...

Person pears = new Person();
pears.age = 32;
pears.weight = 15f;
// ...
```

Figure 7.3: DataLab

Press the "Run" button to view the animation.

Team LIB

PREVIOUS NEXT

Multiple Objects

You have learned that a class is a kind of stencil or cookie cutter for creating objects. Cookie cutters are especially useful if you're going to make lots of cookies. Similarly, classes are really useful because you can use a class to make multiple instances of that class. Different cookies made from the same cutter have the same outline, but they can be frosted or decorated differently. Similarly, different instances of the same class can have different data values.

Figure 7.4 shows three instances of the Person class. Each instance is referred to by a different reference.

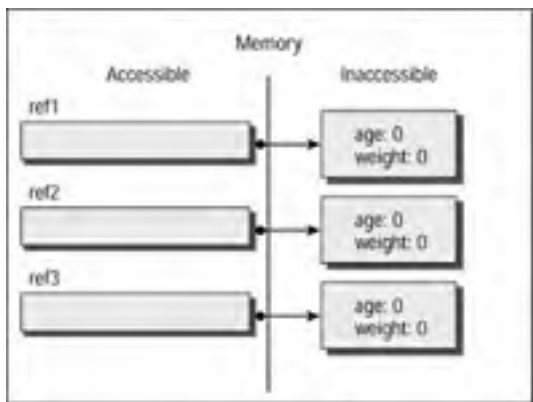


Figure 7.4: Multiple objects

The data values inside an object can be manipulated in all the same ways you can manipulate ordinary variables. For example, if `curly` and `larry` are references to `Person` objects, you might write the following:

```
curly.age = larry.age + 12;
```

The `SeveralObjectsLab` animated illustration lets you play with multiple instances of the `Person` class. Start the program by typing `java objects.SeveralObjectsLab`. You will see the display shown in Figure 7.5.



Figure 7.5: SeveralObjectsLab

`SeveralObjectsLab` initially displays the following code:

```
Person reference1 = new Person();
Person reference2 = new Person();
Person reference3 = new Person();
reference1.age = 30;
reference1.age = 30;
reference1.age = 30;
reference1.age = 30;
reference1.age = 30;
```

The reference variable names aren't very imaginative. Also, it isn't very useful to have five lines that all set the same field in the same object to the same value.

That's where you come in. Type better names into the text fields in the declaration lines. Use the choices to reference different fields in different objects. Use the text fields in the assignment lines to assign any value you like to the fields you have chosen. The assignment values can be literals or expressions, and the expressions can include fields in any of the objects. Figure 7.6 shows `SeveralObjectsLab` after reconfiguring.



Figure 7.6: SeveralObjectsLab reconfigured

Figure 7.7 shows the result of executing Figure 7.7.



Figure 7.7: SeveralObjectsLab reconfigured and executed

Try configuring SeveralObjectsLab to execute the following code:

```
Person simon = new Person();
Person emily = new Person();
Person bethan = new Person();
simon.age = 30;
emily.age = simon.age - 20;
simon.weight = 150;
bethan.weight = simon.weight / 3;
bethan.age = bethan.weight / 5;
```

The program demonstrates construction of the objects and references, followed by assignment to the various fields. Try configuring different values. Hopefully, the results will not be surprising.

The point of SeveralObjectsLab is to get you to think of objects as bundles of data. Objects are similar to other instances of the same class, to the extent that all such objects contain similar clusters of data. That is, each cluster has the same number of variables, and those variables have the same types and names, as defined in the class definition file. However, each object is distinct and has its own version of each variable defined by the class.

Objects and Their Methods

In addition to containing data, objects can also contain methods. You might have suspected as much, because all the application classes presented in this book have contained at least one method (`public static void main(String[] args)`), and sometimes more than one. All those methods have had the keyword `static` in their declarations. Later in this chapter you will see that `static` means, in a sense, "not object-oriented." The methods had to be static because we had not yet introduced objects. Now it is time to present genuinely object-oriented methods.

Let's add a method to the `Person` class:

```
public class Person
{
    int    age;
    short  weight;

    int ageInNYears(int n)
    {
        return age + n;
    }
}
```

The method computes how old the person will be in `n` years. It has a lot in common with the methods you have already seen. It has a declaration that specifies the method's return type, name, and argument list. It has a body enclosed in curly brackets, and it returns a value.

There are two major differences between this method and the ones you have looked at in previous chapters:

- There is no `static` in the declaration.
- The method refers to a field (`age`) of the class where the method is defined.

To call a method of an object, you again use the "reference-dot" notation. The following code shows how this is done:

```
1. Person ed = new Person();
2. ed.age = 62;
3. ed.weight = 220;
4. int n = ed.ageInNYears(3);
5. System.out.println("Ed will be " + n +
    " in 3 years.");
```

Note in line 4 that the method call looks like the method calls you are used to, except that it is preceded by an object reference and a dot. This syntax says, "Call the `ageInNYears` method of the object referenced by `ed`."

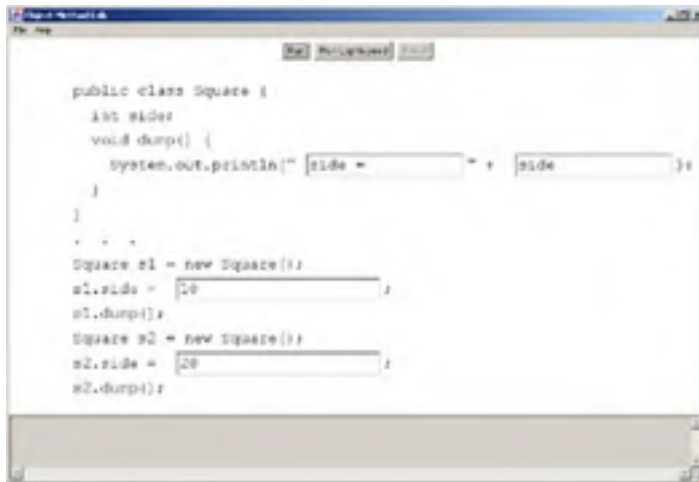
This brings up an important point about object-oriented programming. Until this chapter, all the methods presented in this book have contained in their declarations the mysterious keyword `static`. We will explain `static` in detail in the [next section](#). For now, let's just say that a static method is one that is not object-oriented and does not belong to an individual object. It has been useful to present only static methods for six chapters, because the non-static approach allowed us to introduce a great many foundational concepts without the added complication of presenting objects. But in realistic Java programming, very few methods are static. Most methods are non-static, which means they are associated with objects. Thus, most method calls involve not just invoking a method, but invoking a method *on an object*.

Let's look again at the `Person` class, with line numbers:

```
1. public class Person
2. {
3.     int    age;
4.     short  weight;
5.
6.     int ageInNYears(int n)
7.     {
8.         return age + n;
9.     }
10. }
```

The `ageInNYears` method makes use of the `age` variable. But *which* `age` variable? Every instance of the `Person` class has its own version of `age` (and of `weight`). You may have already guessed correctly: The version of `age` that gets used is the one belonging to the object on which the method call was made. So in the line `int n = ed.ageInNYears(3);`, the version of `age` that gets used is the one belonging to the object referenced by `ed`.

The `ObjectMethodLab` animated illustration demonstrates a class called `Square`. Start the program by typing `java objects.SeveralObjectsLab`. You will see the display shown in [Figure 7.8](#).



```
public class Square {
    int side;
    void dump() {
        System.out.println("side = " + side);
    }
}

Square s1 = new Square();
s1.side = 10;
s1.dump();

Square s2 = new Square();
s2.side = 20;
s2.dump();
```

Figure 7.8: ObjectMethodLab

The class contains one variable, an `int` called `side`. The class also has one method, called `dump`, which outputs a message followed by a value. The output appears in the text area at the bottom of the window. Initially, the method dumps out `side =`, followed by `side` itself. Of course, this is the version of `side` that is owned by the object on which the method was called. The code creates two objects and gives them distinct values for `side`. These values are 10 and 20, but you can change them by typing different numbers into the text fields.

Try configuring the code so that the method dumps the perimeter of the square. Configure again so that the method dumps the area of the square. Observe how, when you call `dump` on an object, the method uses that object's version of `side`.

The [next section](#) will look deeper into how objects contain interacting data and methods.

The Truth About *static*

The time has come to show you how classes, objects, static code, and non-static code all work together to create a complete object-oriented Java application program. To understand what's going on in an application, you need to be clear on the difference between static and non-static parts of a class. You have already learned that methods can be either static or not, and that most methods in most programs are non-static. Data, as well as methods, can be either static or not. Let's look at this distinction.

Static Data

The [previous section](#) explained that when a class defines data members, each instance of the class gets its own version of each data member. This is true for ordinary non-static data. If you add `static` to the declaration of a variable in a class, you defeat this one-version-per-instance mechanism. Instead of getting one version of the variable for each instance, you just get one version of the variable, period.

Here's a version of the `Person` class that has a static variable:

```
1. public class Person
2. {
3.     static int    rev = 3;
4.         int    age;
5.         short  weight;
6.
7.     int ageInNYears(int n)
8.     {
9.         return age + n;
10.    }
11.
12.    void dump()
13.    {
14.        System.out.println("rev " + rev +
15.                            " age = " + age);
16.    }
17. }
```

Static variables have limited uses. One possible use is to keep track of the current revision of the class source. Here you set the `rev` to 3. (As with any other variable declaration, you can initialize a static class variable in the same line where you declare it.) The `rev` is 3 because version 1 from earlier in this chapter just had data, `rev 2` from the [previous section](#) had data and a method, and that brings us to `rev 3`.

When the `dump` method executes, the version of `age` that gets printed out is of course the version belonging to whatever `Person` object is executing the method. The version of `rev` that gets printed out is... well, there is only one version, because the variable is static. Consider the following example:

```
Person thelma = new Person();
thelma.age = 28;
Person louise = new Person();
louise.age = 38;
thelma.dump();
louise.dump();
```

The output of this code is

```
rev 3 age = 28
rev 3 age = 38
```

The (static) `rev` does not change but the (non-static) `age` does.

Now consider the following code, which uses static data to get into trouble:

```
Person thelma = new Person();
thelma.age = 28;
Person louise = new Person();
louise.age = 38;
thelma.rev = 999; // Change thelma's rev
louise.dump();
```

Now the output is

```
rev 999 age = 38
```

If you didn't know that `rev` was static, you would be surprised by the output. The code seems to change the `rev` of `thelma`, not of `louise`. Of course, since `rev` is static, it doesn't belong to an individual object, so it isn't really meaningful to talk about the `rev` "of `thelma`" or "of `louise`." There is just the `rev`.

Java offers you a way to refer to static variables without risking the confusion of the previous example. Instead of saying `thelma.rev` or `louise.rev`, you can say `Person.rev`. In other words, instead of the reference-dot-staticVariableName syntax, you can use `classname-dot-staticVariableName`. This makes static variable usage more conspicuous, because typical class names begin with capital letters, while reference names begin with lowercase letters. (This is not a requirement of the language; it is a style convention. There is no benefit to violating this convention.)

Using the new syntax, the previous example can be rewritten as

```
Person thelma = new Person();
thelma.age = 28;
Person louise = new Person();
louise.age = 38;
Person.rev = 999;
louise.dump();
```

The change makes it clear that the thing that's getting set to 999 is a static variable, so the output should now be no surprise to anyone. The `classname-dot-staticVariableName` notation reinforces the fact that the variable does not belong to any instance. It is convenient to think of statics as belonging to the class as a whole, rather than to an individual instance. By contrast, non-static variables are sometimes called *instance variables*.

We can now move from static data to the more subtle concept of static methods.

Static Methods

You have just seen that a static variable is not associated with an individual object. Similarly, a static method can be thought of as acting in a way that is not associated with an object.

In an instance method (that is, a non-static method), access to an instance variable meant access to the version of the variable owned by the currently executing object. Thus, in the `Person` class, the `ageInNYears` method used the `age` variable. Calling the method on `thelma` meant that the method would use `thelma`'s age. Calling the method on `louise` meant that the method would use `louise`'s age. In all cases, the method used the current object's version of the variable.

In a static method, there is no current object, so it would be meaningless for a static method to use a non-static variable. (Which object's version of that variable should be used? There's no good answer.) A static method is not allowed to read or write the non-static data of its class. Also, a static method may not call the non-static methods of its class.

Sort of.

To really understand what static code can and cannot do, you need to know about a useful Java feature called the *this-reference* notation.

Earlier in this chapter, you learned about the `reference-dot-variableName` and `reference-dot-methodName` notations. These constitute the grammar that lets Java be object-oriented. In object-oriented programming, you specify not only what data or method you want to access, but the object that owns the data or method. But within the instance methods we have seen, instance variables have been accessed without the reference-dot notation. The `ageInNYears` method returned `age + n`, and there is no reference-dot notation there.

In an instance method, any use of a variable without a reference-dot prefix is something like an abbreviation. For example, the method

```
int ageInNYears(int n)
{
    return age + n;
}
```

can be thought of as an abbreviation of

```
int ageInNYears(int n)
{
    return this.age + n;
}
```

The keyword `this`, also known as the *this-reference*, is a reference to the current object. So if you called `thelma.ageInNYears(20)`, within the method, `this` would reference the same object `thelma` referenced.

A static method has no *this-reference*, so it cannot use the abbreviated notation enjoyed by non-static methods. A static method can indeed access non-static data and methods of its own class, or of any other class, but the method must explicitly provide a reference to the intended object. So the following would be perfectly legal:

```
static void printLouisesAge(Person louise)
{
    System.out.println("Louise is " +
        louise.age);
}
```

We can summarize all this static/non-static information as follows:

- A non-static method may use non-static data and methods of its class without using the reference-dot notation. The current object is implied.
- A static method must use the reference-dot notation. There is no current object.
- A static method has no *this-pointer*.

Now at last, we can tie everything together and explain the role of the static `main` method.

The *main* Method

You have seen that static features of a class are a way of getting around the object-oriented requirement that data must live inside objects and methods must be called on objects. Ideally, an object-oriented program would be a federation of objects of many different classes that make method calls on one another, creating new objects as needed and allowing old ones to be garbage-collected when no longer needed. This image is fine once the application is up and running, but how does the process get started? If objects are constructed by other objects invoking `new`, how does the first object get created?

In Java, everything starts with the `main` method. Through the end of the [last chapter](#), you patiently tolerated the presence of `static` in the `main` method's declaration. Now you know what it means: `main` is not called on any individual object. It is static, so

it is just called. Within `main`, objects can be created and non-static calls can be made, so the object interactions quickly become highly object-oriented.

Every application is invoked by typing

```
java ApplicationClassName
```

(The animated illustration programs require a prefix and a dot before the class name. We will explain that notation in [Chapter 9](#).) When you start up an application, a program called `java` is executed. This is the Java Virtual Machine. The JVM does *not* create an instance of the application class. It could have been designed to do so, but the creators of Java decided to let us programmers decide when and how to create objects.

After the JVM initializes itself, it makes use of one of its parts, called the *class loader*. The class loader is code that finds, reads, and interprets `.class` files. After the class loader processes a `.class` file, the JVM is changed in two ways:

- The class defined in the file can be used by the JVM.
- Any static data declared in the class is allocated and initialized.

Initially, the JVM uses the class loader to load the class specified in the command line. Later on, during the course of execution, any class used in the code that has not been loaded already is loaded as needed. Since the class loader allocates and initializes static data before any instances of the class are constructed, you can access a class's static data, and even call its static methods, even if no instances of the class exist.

That's good news, because the next thing the JVM does is call a static method of the class it just loaded. Of course, this is the `main` method. Presumably, `main` constructs objects that construct objects that construct objects, and the program enters its object-oriented phase. Static data and methods can still be used, but typically most accesses are non-static.

The `ObjectLifeCycleLab` animated illustration demonstrates how static code starts the chain of object-oriented interactions. Start the program by typing `java objects.ObjectLifeCycleLab`. At first the display only shows a star, representing the static `main` method of an application. This initial state is shown in [Figure 7.9](#).

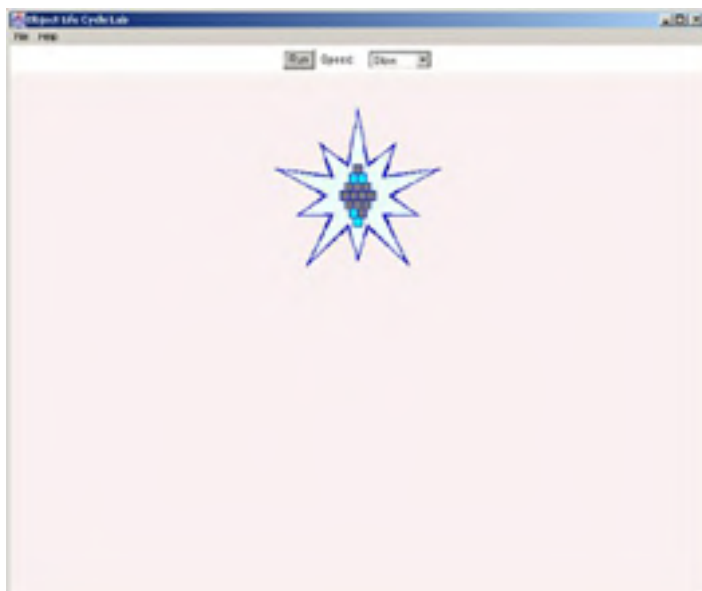


Figure 7.9: `ObjectLifeCycleLab`

When you click the `Run` button, the static code constructs an object that is an instance of one of three classes: `Triangle`, `Rectangle`, and `Oval`. Each object contains its own data and methods. The static code makes a method call on the object, represented by an expanding arrow. The colored dots near the arrowhead represent method arguments.

Now the code in the first object's method constructs a second object, on which a method call is made. The call is returned (that colored dot near the shrinking arrowhead represents a return value), and the life cycle goes on and on and on. Sometimes objects vanish; this represents garbage collection. [Figure 7.10](#) shows `ObjectLifeCycleLab` after running for several minutes.



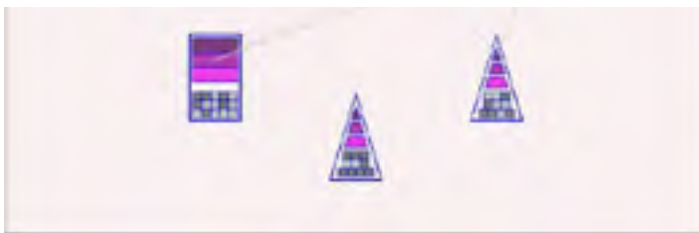


Figure 7.10: ObjectLifeCycleLab after running a while

Of course, ObjectLifeCycleLab is just a symbolic cartoon, but it illustrates several very important concepts. Watch the program until you observe the following behaviors:

- Everything starts with static code, all alone.
- At any moment, the static code or a single object is *current* (recognizable by a highlighted background and flashing data).
- An object only gets garbage-collected if it is not in use.

Reference Data

Variables in a class don't have to be primitive. Classes may define variables that are references to objects or to arrays. Moreover, arrays may contain references rather than primitives. When an object with reference data is constructed, the references are all initialized to the `null` value, indicating that they do not yet point to any objects.

For example, suppose you are writing a Java program to control a weather station that has electronic access to a number of remote thermometers. You might model this situation with two classes, `WeatherStation` and `Thermometer`.

Writing the code that would enable the `Thermometer` class to read input from a physical device is far beyond the scope of this book. Let's just assume that somehow the class has a method called `connect`, which takes care of the connection, and another method called `readTemp`, which returns a float.

The `WeatherStation` class would include the following data declarations:

```
Thermometer[]    therms;
```

As with any other array, the declaration does not create the array. You would create the array in a method, with a line like the following (assuming there are 20 thermometers):

```
therms = new Thermometer[20];
```

Now the array exists, and all its components are `null`. You now have to construct and connect each `Thermometer` object, as follows:

```
for (int i=0; i<therms.length; i++)
{
    therms[i] = new Thermometer();
    therms[i].connect();
}
```

Now you can write a method to compute the average temperature:

```
float getAverageTemp()
{
    float totalTemp = 0;
    for (int i=0; i<therms.length; i++)
        totalTemp += therms[i].readTemp();
    return totalTemp / therms.length;
}
```

Let's refine the `connect` method to illustrate the use of `null`. Sometimes hardware fails. Let's assume that `connect` can detect a failure of the thermometer belonging to the executing object. This failure will be indicated by the method's return value: `true` will mean connection was successful, and `false` will mean there was some kind of failure. You can rewrite the array initialization code like this:

```
for (int i=0; i<therms.length; i++)
{
    therms[i] = new Thermometer();
    if (therms[i].connect() == false)
        therms[i] = null;
}
```

Now you need to refine the `getAverageTemp` method so that it ignores all broken thermometers:

```
float getAverageTemp()
{
    float totalTemp = 0;
    int nWorkingThermometers;
    for (int i=0; i<therms.length; i++)
    {
        if (therms[i] != null)
        {
            totalTemp += therms[i].readTemp();
            nWorkingThermometers++;
        }
    }
}
```

```
    }  
  }  
  return totalTemp / nWorkingThermometers;  
}
```

You use the variable `nWorkingThermometers` to count `Thermometer` objects that actually contributed to the average.

A reference whose value is `null` may not be used for accessing an object. (After all, `null` means that there is no object pointed to by this reference.) For example, the following is illegal:

```
Thermometer thermo = null;  
Float temp = thermo.readTemp();
```

When the second line is executed, the program is terminated abruptly with an error message about a null pointer exception. You have already seen exceptions, but we will not discuss them in detail until [Chapter 11, "Exceptions."](#) The name "null pointer" is a throwback. Java's predecessor languages, C and C++, use pointers, which are like references but less secure. It isn't clear why the exception was named "null pointer exception" rather than "null reference exception."

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Name four traits that arrays and objects have in common.
2. Name two differences between arrays and objects.
3. Objects are not passed as method arguments, but references to objects can be passed. When a reference is passed into a method, any changes made to the referenced object by the method should be visible to the method's caller. Write an application to demonstrate this.

Your application will have two classes: `Cat` and `Ager`. The `Cat` class should have a single variable: an `int` called `age`. The `Ager` class should have a method whose signature is `makeOlder(Cat kitty, int nYears)`. This method should add `nYears` to the age of the `Cat` object referenced by `kitty`. Your `main` method should go in the `Ager` class. It should create one instance of each class, set the cat's age, and then use the `Ager`'s method to change the age. Your `main` should then print out the cat's new age, and verify that it really changed.

4. What happens if you move the `main` method of the previous question from the `Ager` class to the `Cat` class?
5. Write an application that causes a "null pointer exception" failure.
6. What does the following application print out?

```
public class Question
{
    static long x;

    public static void main(String[] args)
    {
        Question q1 = new Question();
        Question q2 = new Question();
        q1.x = 10;
        q2.x = q1.x + 20;
        System.out.println("q1.x = " + q1.x);
    }
}
```

Chapter 8: Inheritance

In the [last chapter](#), you learned that objects are instances of classes, containing data and methods defined by the class. This chapter will present two object-oriented concepts that will greatly enhance what you can do with objects. At first glance, inheritance and constructors do not seem to have much to do with each other. However, by the end of this chapter, you will see that a class's constructors are intimately related to that class's inheritance hierarchy.

Superclasses and Subclasses

You already know that a class has data and methods. You provide a class with these features by writing the code that defines the class. Now it's time to learn another way that a class can get data and methods: *inheritance*.

Inheritance is how a class can get data and methods that are defined in a different class. For this mechanism to work, the two classes must have a special relationship with each other: one must be a *superclass* of the other, which must be a *subclass* of the first. In this section, you'll learn what this relationship means.

Let's start with an example. Suppose you are writing a Java program to support the personnel department of the company. You decide that you should create two classes to represent the employees: `Worker` and `Manager`. These classes have some similarities and some differences. Here are some of the similarities:

- Workers and managers both have employee identification numbers, so both classes have an `int` called `id`.
- Workers and managers both need to get paid, so both classes have a `float` called `salary` and a method called `printCheck`. (The details of creating a method that prints checks are beyond the scope of this book, but it seems only fair that everybody should get a check.)

Now here are some of the differences:

- Managers have workers who report to them, so the `Manager` class has an array of `Worker` objects called `workers`. Workers don't need this, because nobody reports to them.
- Workers might or might not be eligible for overtime pay, so the `Worker` class has a `boolean` called `getsOvertime`. Managers are never eligible for overtime, so the `Manager` class does not need this data field.

Of course, a realistic program would have many more data fields and methods in each class, but this is enough to demonstrate the power and usefulness of inheritance. The `Worker` class looks like this:

```
public class Worker
{
    int    id;
    float  salary;
    boolean getsOvertime;

    void printCheck()
    {
        // Lots of intricate
        // check-printing code
        // goes here.
    }
}
```

And the `Manager` class looks like this:

```
public class Manager
{
    int    id;
    float  salary;
    Worker[] workers;

    void printCheck()
    {
        // Same intricate
        // check-printing code
        // goes here.
    }
}
```

Despite their differences, these classes have a lot in common. The most worrisome common feature is the `printCheck()` method.

Note Notice the empty parentheses after the method name. This is a common practice when writing about a method. It specifies that you're talking about a method rather than a variable or a class.

`printcheck()` is worrisome because it appears in identical forms in two places. Duplication of code should be avoided, because code is never frozen in time. Code evolves. Over the lifetime of a program, bugs are found and new features are required. The process of fixing bugs and adding features is called *maintenance*, and every program requires it. If a method appears in identical forms in two places, every change must be made twice, and the risk of introducing errors rises dramatically.

It is not surprising that workers and managers share some common features. They are both categories of employees. And here we find a simple but profound truth about the way we humans observe our world.

The [previous chapter](#) presented classes as programmatic representations of mental categories, such as "triangle" or "dog."

Object-oriented programming is a very human approach to writing software, because our minds are good at creating categories for the things we experience in daily life. No doubt all animal species do this to some extent, with categories like "food" and "threat" and "safe place to sleep." People do it best of all.

People are so good at creating mental categories that we take the process one step further. We don't just imagine categories of things. With our talent for abstract thinking, we can imagine categories of *categories*! So the "triangle" category is one member of a larger mental concept that we might call "shapes." Other members of this supercategory are "squares" and "rectangles." Similarly, the "dog" category belongs to the supercategory "mammals," which in turn belongs to its own supercategory: "animals."

The Swedish philosopher Carl Linnaeus organized all living species into a hierarchy of supercategories with seven levels. This organization is still in use among biologists. If you've ever had to memorize "kingdom, phylum, class, family, order, genus, species" for a biology class, you were studying Linnaeus' hierarchy. His structure was more detailed than our "animal, mammal, dog" hierarchy. You can't really say that either hierarchy is more or less correct, though. Each one is appropriate for certain tasks.

Well, enough philosophy. The point is that it's natural to think about hierarchies of categories, and Java supports this way of thinking. Let's see how this is done.

Inheritance from Superclasses

In Java, a category is represented by a class. A *supercategory* (if you will continue to permit the use of this made-up word) is represented by a *superclass*. *Superclass* is a real word, and so is its opposite: *subclass*. Every class can have one superclass. That superclass in turn can have its own superclass, and so on. A class may not have multiple superclasses, but multiple subclasses are allowed.

The `extends` keyword is used to denote the superclass/subclass relationship. To see how this works, let's continue the personnel example from the [previous chapter](#). Right before the philosophical digression, you learned that workers and managers are both categories of employees. You will now create an `Employee` class that will contain all the shared functionality of workers and managers.

In Java, every class is capable of being a superclass, and you don't have to do anything special in the class definition of a class that will have subclasses. (The special work, as you'll soon see, comes when you define the subclasses.) So the superclass looks like this:

```
public class Employee
{
    int    id;
    float  salary;

    void printCheck()
    {
        // The same intricate
        // check-printing code
        // goes here.
    }
}
```

Now let's create the `Worker` subclass:

```
public class Worker extends Employee
{
    boolean getsOvertime;
}
```

The class name is followed by the `extends` keyword, which is followed by the class's superclass. That's all we need to do! This works because in Java, there are two ways for a class to have a variable or method:

- The variable or method can be defined in the class.
- The variable or method can be defined in the class's superclass.

This very simple `Worker` class just defines a single variable. But its superclass (`Employee`) defines the variables `id` and `salary`, as well as the method `printCheck()`, so `Worker` also has those variables and that method. We say that `Worker` *inherits* `id`, `salary`, and `printCheck()` from its superclass.

The `Manager` class is also simple:

```
public class Manager extends Employee
{
    Worker[] workers;
}
```

Again, `Manager` *inherits* `id`, `salary`, and `printCheck()` from its superclass. The situation is diagrammed in [Figure 8.1](#).

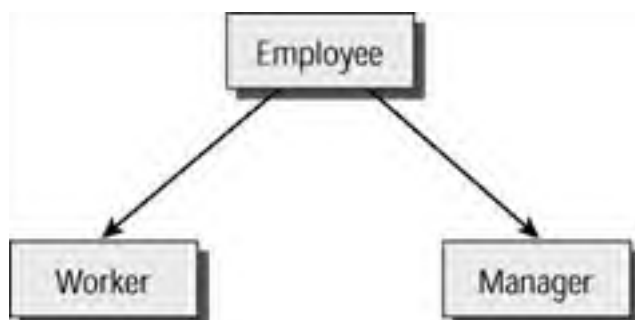


Figure 8.1: A Simple inheritance hierarchy

Figure 8.1 shows that `Employee` is the superclass of both `Worker` and `Manager`. `Employee` itself does not seem to have a superclass, but in Java, every class you define has a superclass, even if you don't explicitly declare one with the `extends` keyword. Java provides a class named `Object`, which is the ultimate ancestor of every class. A class that does not explicitly extend something else extends `Object`.

The Inherit Lab animated illustration lets you create your own class hierarchy diagrams, so that you can see how variables and methods are inherited. To run the program, type `java inherit.InheritLab`. You will see a display that shows a three-level class hierarchy, as shown in **Figure 8.2**.

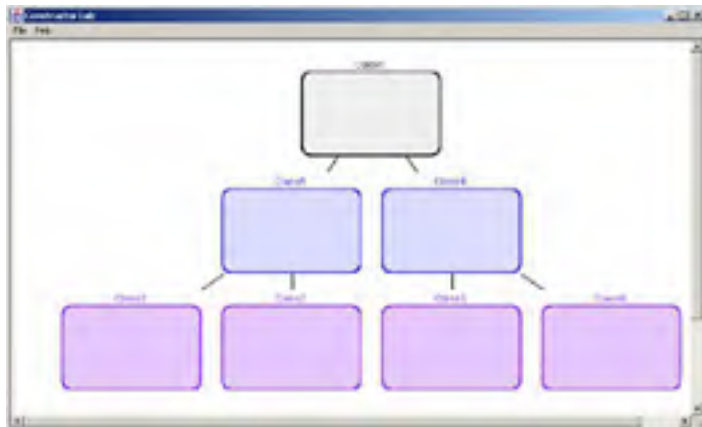


Figure 8.2: Inherit Lab

At the top of the diagram is the `Object` class. `Object` has two subclasses, called `Class1` and `Class4`. Each of those classes has two subclasses. The classes are color-coded based on their level in the hierarchy.

At first the classes are boring. Their names don't mean anything, and they don't have any data or methods. But if you left-click on any class, you'll get a pop-up menu that lets you add a subclass, delete the class, or edit the class. (You can't delete or edit `Object`, since its definition is beyond your control.) First, try adding and deleting classes. Then try editing a class. When you select `Edit` in the pop-up menu, you get a dialog box that lets you change the name of the class, or add or delete data and methods. The dialog box is shown in **Figure 8.3**.



Figure 8.3: Inherit Lab's class-editing dialog box

Try adding a variable to one of the classes in the blue level, just below `Object`. Type a name into the `Add Data` text field, and

then click the Add Data button. Then click Apply. The edit dialog box will go away so that you can see the inheritance diagram. The variable you've added will be seen in the box for the class you edited, and also in all of that class's subclasses, illustrating inheritance of data. You can do the same with methods. Notice that data and methods are color-coded to tell you which class they were defined in.

In the File menu, click on Scenarios. Then look at the two canned hierarchies, which represent animals and transportation. In the Transportation scenario, the bottom-level classes (Car, Bicycle, etc.) inherit from two levels of superclass, as well as from Object. Sophisticated object-oriented programs can have fairly deep hierarchies.

In each scenario, add a subclass at the bottom level and observe the inherited data and methods. Try creating a hierarchy from scratch. If you create something interesting, send us a screenshot or a verbal description at GroundUpJava@sgsware.com. We might include it in the next edition. If so, we'll give you credit.

An Inheritance Example

Let's look at an example of inheritance, expanding on the Worker class from the [previous section](#). Worker is a subclass of Employee, which looks like this:

```
public class Employee
{
    int    id;
    float  salary;

    void printCheck()
    {
        // Whatever.
    }
}
```

Let's add a slightly expanded Worker subclass:

```
1. public class Worker extends Employee
2. {
3.     boolean  getsOvertime;
4.
5.     void dumpSalary()
6.     {
7.         System.out.println("Salary = " + salary);
8.     }
9.
10.    public static void main(String[] args)
11.    {
12.        Worker dagwood = new Worker();
13.        dagwood.salary = 44444.44f;
14.        dagwood.dumpSalary();
15.        dagwood.printCheck();
16.    }
17. }
```

Line 7 of the dumpSalary() method and line 13 of the main() method both act as if salary were an ordinary variable of the Worker class... and they're right. The inherited variables of a class are just like its declared variables. The same is true for inherited methods. Line 15 calls dagwood's printCheck() method, which is inherited.

We will return to this example later on in this chapter. First, it's time to learn what really happens when, as on line 12, an object is constructed.

Construction and Constructors

When you invoke `new` to construct a new instance of a class, automatically a call is made to a piece of code in that class, called its *constructor*. This may come as a surprise, because for the past two chapters you have constructed objects without ever writing constructors, or even knowing about them. In a moment you will see how this works out.

A constructor looks like a method. In fact, there are only two differences between a constructor and a method:

- A constructor has no return type.
- The constructor's name is the same as the class's name.

Like a method, a constructor has a body, enclosed in curly brackets. The code in the body can initialize the newborn object. In fact, this initialization is the basic job of constructors. A constructor can always assume that the object's variables and methods (whether declared in the class or inherited) exist and are accessible for reading, writing, and calling.

Let's add a constructor to the `Worker` class. Assume that all workers in the company have the same salary: \$34,567.89. The `Worker` class (with a different `main()` method) becomes the following:

```
1. public class Worker extends Employee
2. {
3.     boolean getsOvertime;
4.
5.     Worker()
6.     {
7.         salary = 34567.89f;
8.     }
9.
10.    void dumpSalary ()
11.    {
12.        System.out.println("Salary = " + salary);
13.    }
14.
15.    public static void main(String[] args)
16.    {
17.        Worker dagwood = new Worker();
18.        dagwood.dumpSalary();
19.    }
20. }
```

The constructor is lines 5-8. Constructors, variables, and methods can appear in any order in the body of a class definition. However, it is common practice to have variables come first, followed by constructors, followed by methods. Usually, the `main()` method comes at the end. When this application is run, line 17 constructs a new instance of `Worker`. First, space for all variables is allocated. Then the constructor code is called. Line 7 of the constructor initializes `salary` to 34567.89, so the call to `printSalarydumpSalary()` prints out `Salary = 34567.89`.

Overloading Constructors

It is not especially realistic to expect every worker in a company to have the same salary. Fortunately, you can pass arguments into a constructor the same way you pass them into a method. As with a method, you can put an argument list inside the parentheses that follow the constructor name. Those arguments are accessible within the method. The next version of `Worker` has a constructor that accepts an argument.

```
1. public class Worker extends Employee
2. {
3.     boolean getsOvertime;
4.
5.     Worker(float sal)
6.     {
7.         salary = sal;
8.     }
9.
10.    void dumpSalary ()
11.    {
12.        System.out.println("Salary = " + salary);
13.    }
14.
15.    public static void main(String[] args)
16.    {
17.        Worker dagwood = new Worker(55555.55f);
18.        dagwood.dumpSalary();
19.    }
20. }
```

The constructor now takes a float argument, and the invocation on line 17 passes a float. The output is `Salary = 34567.89`.

In [Chapter 4, "Methods,"](#) you learned that methods are polymorphic. That is, different methods within a class may share a common name, as long as their argument lists are different. The practice of reusing a method name in a class is called *overloading*. (The term has a negative connotation in real life. When people or bridges are overloaded, that's bad. But in programming, there is nothing bad about overloading.) You can also overload a class's constructor so that the class has multiple constructors, as shown here:

```
public class Manager extends Employee
{
    Worker[] workers;

    Manager(int nWorkers)
    {
        workers = new Worker[nWorkers];
    }

    Manager(float sal, int nWorkers)
    {
        salary = sal;
        workers = new Worker[nWorkers];
    }
}
```

This class has two constructors. In both versions, you specify the number of workers. In the second version, you also specify the manager's salary.

Default Constructors

This section answers an important question: In the [previous chapter](#) and the first part of this one, how was it possible to construct objects in classes that didn't have any constructors? To understand the answer, you have to know what a *no-args constructor* is. It's just a constructor with an empty argument list.

When you create a class with no constructors, the compiler creates a no-args constructor automatically. A no-args constructor that is created automatically is called a *default constructor*. You only get a default constructor if your class does not explicitly have any constructors. If your class has constructors, no matter how many, no-args or otherwise, no default constructor is created for you.

This mechanism assures that every class has at least one constructor, even if the class was written by someone who has never heard of constructors!

A default constructor does almost nothing. It contains no initialization code, because it contains no code at all. All it does is participate in the constructor chain mechanism, which is discussed in the [next section](#).

The Chain of Constructions

Objects are like onions. They consist of layers within layers within layers. Consider the `Submarine` class from InheritLab's Transport scenario. This class extends `WaterTransport`, which extends `Transport`, which extends `Object`. One way to visualize an instance of `Submarine` is as an instance of `Object` forming an inner core. Around this core is a layer consisting of the data and methods of an instance of `Transport`. In turn, this layer is surrounded by a `WaterTransport` layer, which is surrounded by a `Submarine` layer. The layered structure is shown in [Figure 8.4](#).

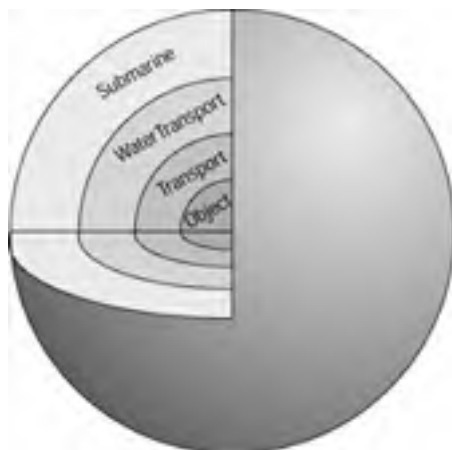


Figure 8.4: Object layers

When a `Submarine` instance is constructed, each layer's constructor is called in turn, starting with `Object` (the innermost layer) and moving outward. This mechanism doesn't have an official name, but we'll call it the *chain of constructors*.

The ConstructorLab animated illustration shows the chain of constructors in action. Start the program by typing `java inherit.ConstructorLab`. At first glance, the program looks just like the InheritLab program that you already saw earlier in this chapter. But when you click on an object, the pop-up menu has an extra Construct... item. If you make this selection, you see an animation of the layer-by-layer construction of the class you've selected. Try it with the `Submarine` class from the Transport scenario.

Here's how the chain-of-constructors mechanism works: When a constructor is called, but before any of its code is executed, a call is made to the no-args constructor of the superclass. If the superclass is not `Object`, before any of its own constructor code is executed, a call is made to its own superclass's no-args constructor. This chain of calls continues up the inheritance hierarchy until it reaches `Object`, which has no superclass.

So when you call a constructor for `Submarine`, the first thing that happens is a call to the no-args constructor for `WaterTransport`. Within that constructor, the first thing that happens is a call to the no-args constructor for `Transport`. Finally, within `Transport`'s no-args constructor, a call is made to the `Object` no-args constructor. At this point, the chain ends.

Why does Java do this? Consider the benefits if you are the person who writes the `Submarine` class code. Your constructors might need to access the data of the `WaterTransport` superclass, which might be initialized by `WaterTransport`'s constructor. That constructor might need to access the data of its own superclass, and so on. The chain-of-constructors mechanism guarantees that by the time a class's constructor code begins to execute, the superclass portion of the class is intact and valid.

Note Bear in mind that the mechanism operates without your having to do anything at all. You don't have to make it happen. As a matter of fact, you can't avoid it. But you can slightly alter its behavior.

As you know, constructor invocation begins with an automatic call to the superclass's no-args constructor. Recall that constructors can be overloaded so that alternate superclass constructors could be available. There are two reasons why you might want to invoke a different superclass constructor:

There is no superclass no-args constructor. This happens if you explicitly give the superclass one or more constructors that take arguments. Now you don't get an automatic default constructor, so unless you explicitly coded a no-args constructor for the superclass, there won't be one.

The superclass has a no-args constructor that doesn't do what you want. But there's another constructor that does exactly what you want.

In either of these situations, you still want the construction chain to happen. You just want to invoke a different version of the superclass constructor. This is done with the `super` keyword.

To see how `super` works, let's extend the `Manager` class from earlier in this chapter. The class looks like this:

```
public class Manager extends Employee
{
    Worker[] workers;

    Manager(int nWorkers)
    {
        workers = new Worker[nWorkers];
    }

    Manager(float sal, int nWorkers)
    {
        salary = sal;
        workers = new Worker[nWorkers];
    }
}
```

This class provides its own constructors, so there is no default constructor provided by the compiler. Neither is there an explicitly coded no-args constructor, so any subclass of `Manager` will have to modify the construction chain to avoid invocation of a constructor that doesn't exist.

Let's create a subclass called `Officer`. An officer is a high-ranking manager who may or may not serve on the board of directors. The subclass will have an `int` variable called `nYrsOnBoard`, which tells how many years (if any) this officer has served on the board. There will also be a single constructor whose arguments are the number of workers reporting to this officer and the initial value for `nYrsOnBoard`. Officer salaries are \$850,000.00. Nice work if you can get it. The `Officer` code looks like this:

```
1. public class Officer extends Manager
2. {
3.     int nYrsOnBoard;
4.
5.     Officer(int nWorkers, int initialNYrs)
6.     {
7.         super(850000f, nWorkers);
8.         nYrsOnBoard = initialNYrs;
9.     }
10. }
```

The line to notice is line 7, which introduces the `super` keyword. It looks like a call to a method called `super()`. In fact, the code in the parentheses is an argument list, but you aren't allowed to create a method with that name. Instead, `super` is a signal that you are modifying the construction chain by requesting a call to a different superclass constructor (that is, one that isn't the no-args version). This use of `super` may only appear as the first executable code in a constructor (so if you reversed lines 7 and 8, you would get a compiler error).

Line 7 invokes an alternative superclass constructor, but which one? This is determined by the argument list inside the parentheses that follow `super`. Here there are two values: a float followed by an `int`. So the `Manager` constructor that gets invoked will be the version that takes two arguments: a float and `int`. If you look at the `Manager` constructor, you'll see that this corresponds to the second of its constructors.

Overriding

You have already seen method overloading, where a method name is reused in a class definition. Java also supports method *overriding*, which looks like it ought to be similar to overloading because the spellings are so similar. In fact, overriding is indeed a kind of method name reuse. In this case, the reuse is within an inheritance hierarchy.

To continue this ongoing example, the `Officer` class is the bottom member of a three-level hierarchy. This is shown in [Figure 8.5](#).



Figure 8.5: Inheritance of `Officer`

In the original version of `Employee`, you imagined a `printCheck()` method. This method is inherited by `Worker`, `Manager`, `Officer`, and any other subclasses of `Employee`, immediate or indirect, that you might create in the future.

But what happens if a class inherits a method that is not appropriate to that class's operation? For example, `printCheck()` might print checks on a monochrome printer that is loaded with low-cost blank check forms. That's fine for ordinary workers, and even for managers, but officers need to have their checks printed by a special color printer, on high-grade fancy paper. So the inherited version of `printCheck()` just won't do.

The solution is to override `Officer`'s inherited version of `printCheck()`. You do this simply by putting into the `Officer` code a version of `printCheck()` that does what you want it to do. The code looks like this:

```
1. public class Officer extends Manager
2. {
3.     int nYrsOnBoard;
4.
5.     Officer(int nWorkers, int initialNYrs)
6.     {
7.         super(850000f, nWorkers);
8.         nYrsOnBoard = initialNYrs;
9.     }
10.
11.     void printCheck()
12.     {
13.         // Whatever, and make it fancy.
14.     }
15.
16.     public static void main(String[] args)
17.     {
18.         Officer julius = new Officer(25, 50);
19.         julius.printCheck(); // Fancy
20.         Worker dagwood = new Worker(44444.44f);
21.         dagwood.printCheck(); // Plain
22.     }
23. }
```

Again, we've left out the details of how to print a check. The import point is line 11, where you have a declaration of `printCheck()` that looks just like the version in `Employee`. The declarations are the same, but the bodies are different. Look at the `main()` code. On line 19, you call `printCheck()` on an officer. The overriding (fancy) version of the method will be called. On line 21, you call `printCheck()` on a worker. Since `Worker` does not override the method, the inherited (plain) version is called.

In order for one method to override another, the superclass version and the subclass version must have identical return types, method names, and argument list types. It is illegal for the return types to be different if the method names and argument list types match. If the names or argument lists are different, that's legal but it isn't overriding.

Overriding allows you to use a very powerful kind of polymorphism, which will be explained in the [next section](#).

Polymorphism Revisited

Recall from [Chapter 4](#) that polymorphism means "many forms," implying "one name, many forms." In [Chapter 4](#), you saw polymorphism involving method overloading (reusing a method name within a class). The [previous section](#) presented overriding, which is polymorphism of a different kind: *name reuse in an inheritance hierarchy*.

This section will present a powerful technique involving overriding polymorphism. Let's begin by exploring the difference between the class of an object and the type of a reference.

The only way to create an object is to call a constructor. Not surprisingly, an object's *class* is the class whose constructor was called when the object was created. So `new Officer(50, 3);` creates an object whose class is `Officer`.

A reference is not an object. You have already seen that a reference is a configuration of 32 bits that uniquely identifies an object. References, not objects, are passed as method arguments. References, not objects, are declared as variables. The *type* of a reference variable is the type that appears in the variable's declaration. So `Worker dagwood;` declares that `dagwood` is a reference of type `Worker`.

So far, this should be glaringly obvious. Almost always, when a reference points to an object, the type of the reference is the same as the class of the object. This happens, for example, in the line `Worker dagwood = new Worker(22222.22f);`.

But sometimes the reference type and the object class are not the same. Let's introduce this idea with an analogy to something you already know about. In [Chapter 3, "Operations,"](#) you learned that you can assign a numeric value to a variable whose type is the same as, or wider than, the type of the numeric value. So you can assign a byte to a float, or an int to a double, as shown here:

```
byte b = 12;
float f = b;
int i = 54321;
double d = i;
```

The new type (on the lhs of the assignment) must have enough capacity to encompass the value. Any extra capacity is no problem. The same rule holds when you're passing an argument to a method. If the method expects an argument of a certain type, you can pass data of any type, provided the type declared in the method is the same as, or wider than, the type you actually pass.

A similar principle applies when you're assigning references. Consider this code:

```
newRef = oldRef;
```

It is legal for `newRef` and `oldRef` to have different types, as long as the type of `newRef` is above the type of `oldRef` in the inheritance hierarchy. So the following is legal:

```
1. Worker dagwood;
2. Employee emp;
3.
4. dagwood = new Worker(22222.22f);
5. emp = dagwood;
```

Here you have one object with two references. Clearly the class of the object is `Worker`, because it is `Worker`'s constructor that is called on line 4. On line 5, the type of reference `emp` is `Employee`, which is the immediate superclass of reference `dagwood`. Thus, the rule is obeyed, and line 5 is legal. You could also pass `dagwood` as an argument to a method that declared it took an `Employee` argument.

So now you know that the type of a reference can be different from the class of the object the reference points to. This puts a subtle but very important restriction on the Java compiler. You and I can look at lines 4 and 5 and say to ourselves, I know `emp` has type `Employee`, but really it points to a `Worker`. *We* can do that, but the *compiler* can't.

This isn't a shortcoming on the part of the people who wrote the compiler. In fact, the developers of the Java compiler are some of the smartest programmers in the world. But there are fundamental theoretical limits on what a compiler can do. No car designer, no matter how brilliant, can make a race car that goes faster than light. Similarly, nobody can create a compiler that, in the general case, can look at a reference and know what the class of the reference's target will be when the code is executed.

To put this more succinctly: *At compile time, the compiler only knows the types of references. It does not know the classes of objects.* This means that a reference's type dictates the variables and methods you can access via that reference. You may only access variables and methods that are the same type as the reference. To return to our example, the reference `emp` has type `Employee`, so by using `emp`, you can read and write variables and call methods of `Employee`. (The data and methods can be implemented in the `Employee` class, or they can be inherited from a superclass.) Using the reference `dagwood` (whose type is `Worker`), you can access the data and methods (whether directly implemented or inherited) of `Worker`. This is shown in [Table 8.1](#):

Table 8.1: References, Variables, and Methods

Variables via emp	Variables via dagwood	Methods via emp	methods via dagwood
Id	Id	printCheck()	printCheck()
salary	salary		dumpSalary()
	getsOvertime		

Given the information in [Table 8.1](#), the previous code example might be baffling. Here is the code:

```
1. Worker dagwood;
2. Employee emp;
3.
4. dagwood = new Worker(22222.22f);
5. emp = dagwood;
```

The table clearly shows that the reference `dagwood` gives you access to all the data and variables you can get to via `emp`, and more. Even though line 5 is legal, why would you ever do it? Line 5 just trades a perfectly good reference for one that is less powerful. There must be some compensating benefit to doing this, or nobody in their right mind would ever want to.

In fact, there is a very valuable compensating benefit, as you will see in the [next section](#).

Inheritance Polymorphism

Let's continue our code example just a bit further. No doubt, there must be a piece of code somewhere in the program that periodically prints a paycheck for everybody who works at the company. Let's suppose there's a class called `Paymaster` that knows who all the employees are. `Paymaster` might look something like this:

```
public class Paymaster
{
    Worker[]    workers;
    Manager[]   managers;
    Officer[]   officers;

    void payEveryone()
    {
        for (int i=0; i<workers.length; i++)
        {
            Worker wor = workers[i];
            wor.printCheck();
        }
        for (int i=0; i<managers.length; i++)
        {
            Manager man = managers[i];
            man.printCheck();
        }
        for (int i=0; i<workers.length; i++)
        {
            Officer off = officers[i];
            off.printCheck();
        }
    }
}
```

In reality, the `Paymaster` class would need a lot more code, including a constructor to set up the three arrays. In fact, there would be a lot more arrays. Companies don't just have workers, managers, and officers. They have presidents, vice presidents, directors, part-timers, and possibly many others. There could be lots of categories of people who need to get paid, and if there had to be one array for each category, that would make for a lot of arrays.

Just to hammer the point home, let's suppose there are classes called `President`, `VP`, `Director`, and `PartTimer`, each of which extends `Employee`. We won't show the code for these classes, but here is the monster that `Paymaster` has become:

```
public class Paymaster
{
    Worker[]    workers;
    Manager[]   managers;
    Officer[]   officers;
    President   prez;    // No array: there's only 1 president
    VP[]        vps;
    Director[]  directors;
    PartTimer[] partTimers;

    void payEveryone()
    {
        for (int i=0; i<workers.length; i++)
        {
            Worker wor = workers[i];
            wor.printCheck();
        }
        for (int i=0; i<managers.length; i++)
        {
            Manager man = managers[i];
            man.printCheck();
        }
        for (int i=0; i<workers.length; i++)
        {
            Officer off = officers[i];
            off.printCheck();
        }
        prez.printCheck();
        for (int i=0; i<vps.length; i++)
        {
            VP veep = vps[i];
            veep.printCheck();
        }
        for (int i=0; i<directors.length; i++)
        {
```

```
        Director dir = directors[i];
        dir.printCheck();
    }
    for (int i=0; i<partTimers.length; i++)
    {
        PartTimer pt = partTimers[i];
        pt.printCheck();
    }
}
}
```

Let's see how much you can simplify this code, using inheritance polymorphism. The first thing to do is eliminate all those arrays and replace them with a single array, called `employees`:

```
public class Paymaster
{
    Employee[] employees;
    . . .
```

The components of the new array aren't really employees. That is, `employees` is an array of references whose types really are `Employee`, but the classes of the objects pointed to by those references are really `Worker`, `Manager`, `Officer`, and so on. The array is initialized by a lot of code along the following lines, which might appear in `Paymaster`'s constructor:

```
. . .
Worker      dagwood;
Manager     julius;
President    preston;
Director    deirdre, dirwood;

. . .

employees[1154] = dagwood; // Employee <- Worker
employees[1155] = julius;  // Employee <- Manager
employees[1156] = preston; // Employee <- President
employees[1157] = deirdre; // Employee <- Director
employees[1158] = dirwood; // Employee <- Director

. . .
```

The `employees` array is a cluster of references, all of type `Employee`. Each of the 5 commented assignment lines stores a reference in a component of the array, and not one of those references is actually of type `Employee`. That's okay. The rhs references are all of types that are subclasses of `Employee`, so the "up-the-inheritance-hierarchy" assignment rule is obeyed.

Now let's return to `Paymaster`'s `payEveryone()` method. Here is all you have to do:

```
void payEveryone()
{
    for (int i=0; i<employees.length; i++)
    {
        Employee emp = employees[i];
        emp.printCheck();
    }
}
```

That's all! All the references to all the people are now living peacefully together in one diverse community... er, array of references, where the classes of the objects pointed to are unknown and mixed. But you do know that every class is a subclass of `Employee` (or is `Employee` itself). So every object has a `printCheck()` method. This method might be the version inherited from `Employee`, or it might be an overriding version.

What happens when the `payEveryone()` loop pays an officer? Recall that the `Officer` class overrides `printCheck()` to use a fancy printer with fancy paper. You have a reference (some component of the `employees` array) of type `Employee`, pointing to an object of class `Officer`. Each has its own version of `printCheck()`. Which one wins?

The answer, and this is crucially important, is that *the type of the reference is ignored*. The class of the object being called determines which version of an overridden method will be called. So in this example, all the officers will get their checks printed in fancy paper, and any other classes that override `printCheck()` will have the appropriate version called.

In case this is overwhelming you, let's look at a very simple example that illustrates the same principle:

```
public class FlyingMachine
{
    void whoAreYou()
    {
        System.out.println("I am a flying machine.");
    }
}
```

Subclass `FlyingMachine` like this:

```
public class Helicopter extends FlyingMachine
{
    void whoAreYou()
    {
        System.out.println("I am a helicopter.");
    }

    public static void main(String[] args)
    {
        FlyingMachine fm = new Helicopter();
        fm.whoAreYou();
    }
}
```

When the application runs, the output is "I am a helicopter." This proves that the class of the object, not the type of the reference, determines the method version that gets called.

Team LIB

← PREVIOUS NEXT →

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Which of the following hierarchies illustrate a good understanding of the difference between classes and objects? Which ones represent mistaken understanding? The arrows mean "has subclass", so in option A, Shape → Triangle means "class Shape has subclass Triangle".

1. Shape → Triangle → RightTriangle
2. GreatLiterature → GreatPoem → DivineComedy
3. Planet → Continent
4. Person → HeadOfState → Emperor
5. Person → HeadOfState → Emperor → AugustusCaesar

2. Which of the following classes have a no-args constructor?

1. A)

```
class A { }
```
2. B)

```
class B
{
    B() { }
}
```
3. C)

```
class C
{
    C(int x) { }
}
```
4. D)

```
class D
{
    D(int y) { }
    D() { }
}
```

3. Write the code for two classes. The first, called `WaterBird`, has a float variable called `weight`. The class has a single constructor that looks like this:

```
WaterBird(float w)
{
    weight = w;
}
```

Compile this class. Now create the second class, called `Duck`, which extends `WaterBird`. `Duck` has no variables or methods, so it shouldn't take you long to write it. Will `Duck` compile? First, think about the issues involved. Then try to compile `Duck` and see if you were right.

4. Write some code to demonstrate to yourself the chain of construction. Create an inheritance hierarchy of 4 classes. Give them any names you like. They don't have to have any data or methods, but each one should have a no-args constructor. These constructors should print out a line identifying the current class (something like "Constructing an instance of `WaterBird`"). Your `main()` method should construct a single instance of your lowest-level subclass. What is the output? Does it matter which class contains the `main()` method?
5. Write some code to demonstrate inheritance polymorphism. Create a superclass class with 3 subclasses. The superclass should have a method that prints out a line identifying the current class (something like "I am a Monster"). Two of the subclasses should override this method to print out a different message (like "I am a Werewolf"). Give the superclass a `main()` method with an array of size 4, typed as the superclass (for example, `Monster[] monsters = new Monster[4];`). Your `main()` should populate the array with references to 4 objects, each with a different class, and then traverse the array, calling your method on each array component. What is the output? Does it matter which class contains the `main()` method?

Chapter 9: Packages and Access

Overview

Congratulations! At this point, you know almost all there is to know about Java's classes. The remainder of this book will look at how classes interact, and it will present many of the core Java classes that the system provides to make your life easier. To make an analogy with the life sciences, we are pretty much done with class anatomy (the analysis of the internal structure of a class) and are ready to tackle class sociology (the study of how classes interact).

This chapter will first look at packages, which are organizations of interrelated classes. Once you understand packages, you will have a good foundation for understanding access. Java has several keywords that control access, including `public`. This means that by the time you finish this chapter, you will know why your application classes and your main methods have been marked as `public`.

By the way, in this edition of this book, this chapter has no animated illustrations. The information presented here doesn't benefit from the animated illustration model. However, if you can think of a concept from this chapter that would look good as an animation, please send us your idea in detail at groundupjava@sgsware.com. If we use your idea in the next edition, we will give you a credit.

Packages

A *package* is a named group of classes. Generally, the classes of a package are collected together into a directory. It is possible for package classes to appear in more than one directory, but it's hard to imagine when this would be helpful. So a package looks a lot like a directory, even though they aren't exactly the same thing.

There is another similarity: Just as a directory can contain files and subdirectories, a package can contain classes and subpackages. For example, a package called `acmeproducts` might contain two classes named `Database` and `Connection`. The package might also contain a subpackage called `utilities`, which contains three classes named `ThreadPool`, `Mailbox`, and `UserProfile`. Your package structure would most likely appear in the directory structure shown in [Figure 9.1](#).

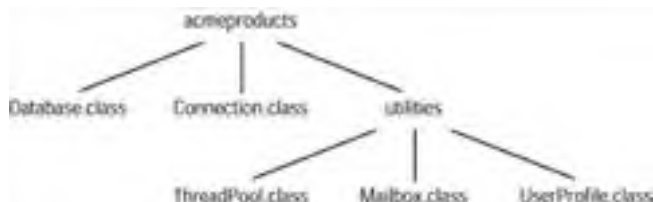


Figure 9.1: Example package/ directory structure

It's important to realize that the directory and the subdirectory shown in [Figure 9.1](#) are not the actual package and subpackage. They are just the places where the classes of the package and subpackage are found. Soon you'll learn how to put your own classes in packages, and why you might want to do so. For now, be aware that there is more to it than just creating the right directory structure and storing the class files appropriately.

When a class is part of a package, the class has a long, formal, official name. It's important to understand this long name, even though it's rarely used. The official name of a class consists of its package structure, from top to bottom, followed by the class name as defined in the source file. All these elements are separated by periods. For example, the `Mailbox` class in [Figure 9.1](#) would be defined in a file called `Mailbox.java`. Its full name is `acmeproducts.utilities.Mailbox`, because it lives in subpackage `utilities`, which lives in package `acmeproducts`. (Note that the package name is all lowercase. You are allowed to use uppercase in package names, but by convention, nobody does.)

There is yet another parallel between directories and subpackages. Even a modest laptop can have tens of thousands of files on its hard drive. If every file on the drive had to have a unique name, keeping track of which names were still available would be a horrendous task. Thanks to directories, you only have to maintain name uniqueness within directories. So you might have a directory called `photos`, with a subdirectory called `NewYearsParty`, which contains a file called `JulieAndRich.jpg`. If you had a housewarming party, you could create a subdirectory of `photos` called `housewarming`, and if you took a picture of Julie and Rich at the housewarming, you could store it in the `housewarming` subdirectory under the name `JulieAndRich.jpg`.

We say that a directory structure provides a *namespace*. A namespace is a way of organizing resources (files, classes, etc.) so that name uniqueness only has to be maintained in relatively small and manageable regions.

Packages are also namespaces. Within a package, all class names must be unique. However, names may be reused in different packages without restriction. [Figure 9.2](#) shows a package structure that might be used by a fictional company called Stained Glass Software. This company has two product lines: database products and ray-tracing software.

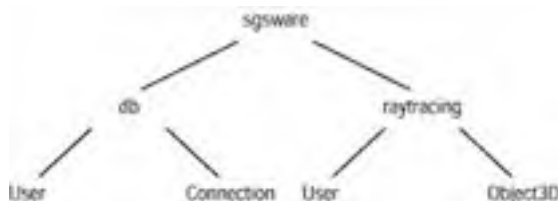


Figure 9.2: Package as namespace

Stained Glass Software has dozens of programmers, working in two divisions on opposite sides of the world. Life would be impossible if every programmer had to check with every other programmer before creating a new class, just to make sure the class name wasn't already in use. As you can see from [Figure 9.2](#), both divisions have created a class called `User`. Fortunately this isn't a problem, because the two classes are in different packages, and packages are namespaces. To put this another way, one class is really called `sgsware.db.User`, and the other is really called `sgsware.raytracing.User`. You can see that packages support collision-safe naming. In the real world, a package might be developed and maintained by several workgroups, by a single workgroup, by a few individuals, or by a single person.

Creating Your Own Package

By now, we hope you're convinced that packages are a good thing. Here's how to create your own packaged classes.

You need to do two things:

- Use the `package` keyword in your class source code.
- Compile with the `-d` flag.

It's interesting to think about what *isn't* in this list. Here's what you *don't* have to do:

- Create a package.
- Create the package directory structure.
- Move the class files into the directory structure.

Let's suppose you are the founder of Stained Glass Software. You have a new computer, fresh out of the box, and you are ready to write the `sgsware.raytracing.Object3D` class. This is the company's first class, so no structure has been created yet.

The first step is to create a directory where the package structure will go. Let's say you decide to put it in `/products/revA`. (You might be using the kind of computer that uses the backslash as a file path separator, but for simplicity we'll use forward slashes throughout this book.) After making sure that `/products/revA` exists, you create a directory to hold your source code. We'll call it `/products/source/ray`. So initially, your directory structure looks like [Figure 9.3](#).

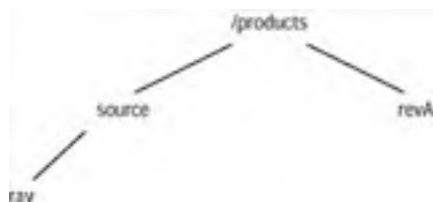


Figure 9.3: Initial directory structure

When you write your source code, you have to tell the compiler that the class belongs to the `sgsware.raytracing` package. To do this, you use the `package` keyword. Besides comments, the package declaration must be the first code in your source file:

```
// This class belongs to a package.
package sgsware.raytracing;

public class Object3D
{
    . . .
}
```

When you compile, use the `-d` command line option. This should come after `javac`, and it should be followed by a space. After the space comes the directory where the package structure is to be stored. Since you're putting your package of classes in `/products/revA` (known as the *destination directory*), you would compile like this:

```
javac -d /products/revA Object3D.java
```

The destination directory must exist before the command is typed. The compiler realizes that the class file should be `/products/revA/sgsware/raytracing/Object3D`. The compiler will create any required subdirectories in the destination directory, and it will place the class file it generates in the appropriate place. So after compilation, your directory structure would look like the one shown in [Figure 9.4](#).

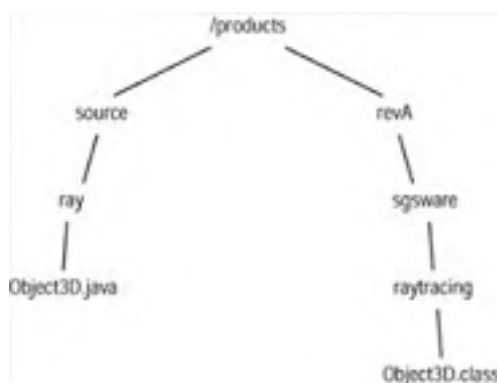


Figure 9.4: After compilation

When you want to create a second `raytracing` class, you can put your source file anywhere you like. However, if you put the source file anywhere other than `/products/source/ray`, it will only make life more baffling for yourself and others. Again, your source should declare that it belongs to the `sgsware.raytracing` package:

```
package sgsware.raytracing;

public class User
{
    . . .
}
```

When you compile, you again use the `-d` flag:

```
javac -d /products/revA User.java
```

This time, the compiler does not have to create a subdirectory for storing the generated class file, since that subdirectory already exists. After compilation, your directory structure looks like [Figure 9.5](#).



Figure 9.5: After more compilation

Now that you know how to create your own packages, let's look at how to use them.

Finding Packages: *classpath*

The designers of Java assumed that you would be working in an environment where you would be using lots of different packages. This was a safe assumption, since Java itself comes with lots of different packages that support functions like string manipulation, file I/O, and GUI components. Moreover, you are likely to be using other packages that you've created yourself, that are standard for your company, or that were bought from a third party. You might be developing code that uses classes from many different packages. When you compile and execute, the compiler and JVM need to know where in your file system these custom packages are located. You do this with the *classpath*.

The classpath is a list of directories, or *classpath elements*, that contain package structures. The classpath elements can be specified in two places:

- The CLASSPATH environment variable
- The -classpath or -cp argument of the javac or java command line

An environment variable is a variable whose scope is your computing session, rather than the interior of a program. Individual programs can read environment variables and take action accordingly. The CLASSPATH variable, which is read by both the Java compiler and JVM programs, is a list of classpath elements, separated by semicolons (;) for Windows machines and by colons (:) for other systems.

You can set CLASSPATH either by typing a command or by running a script. Running a script is easier (after you create the script). Some people prefer to set CLASSPATH in their boot or login scripts (or whatever the equivalent is on their own machines). Appendix A, "Installing Java" shows how to write a script that sets CLASSPATH to ".", which is the current working directory. Different operating systems use different commands to set an environment variable, as detailed in the appendix.

The other way to specify classpath elements is to type them into your compilation and execution command lines. You do this after the javac or java command: type `-classpath`, then a space, then the classpath elements you want to specify. As with CLASSPATH, if you have more than one element, they should be separated by semicolons (;) for Windows machines and by colons (:) for other kinds of machines.

Suppose you are using a Windows machine, and you have acquired and built lots of custom packages. These packages are stored in three different directories: `\a\b`, `\c\d`, and `\e\f`. There might be more than one package in any of these directories. The names aren't very creative, and three is an inconveniently large number of classpath elements, but it makes for a nice clear example.

Now suppose that, for some reason, you want to specify `\a\b` and `\c\d` in CLASSPATH, while specifying `\e\f` on the command line. You would start by setting CLASSPATH (either manually or in a script) as follows:

```
SET CLASSPATH=\a\b;\c\d
```

Then you would compile source like this:

```
javac -d \my\destination -classpath \e\f MyThing.java
```

If MyThing is your application class, and it lives in package `sgsware.db`, you would execute your application like this:

```
java -classpath \my\destination;\e\f sgsware.db.MyThing
```

The application class, as well as any other classes used at any time in the application, must appear in one of the classpath elements. So the MyThing.class file must be in one of the following directories:

- `\a\b\sgsware\db`
- `\c\d\sgsware\db`
- `\e\f\sgsware\db`

Note that each of these directories consists of a classpath element, followed by a package structure.

The Java compiler and the JVM use a piece of code called a *class loader*. The class loader finds class files, reads them, and translates them into internal representations. The first step in the process is to find files. The class loader does this by looking in each classpath directory in turn. In each directory, it looks for a subdir that corresponds to the package of the class being loaded. So when the class loader looks for MyThing.class, it looks in each of the directories listed in the preceding bulleted list.

Now you know how to create, store, and use packages. But there is still a problem, which will be presented and solved in the [next section](#).

Importing

You have seen that a class name is really a list of package elements, separated by periods and ending with the (short) name of the class. Packages provide a convenient way to organize software and reduce naming headaches, but there seems to be a tradeoff with what happens in your source code.

In the [previous section](#), you considered a class called `MyThing` in a package called `sgsware.db`. The true name of class `MyThing` is `sgsware.db.MyThing`. That doesn't seem so bad until you realize that the following line of source code is not allowed:

```
MyThing m = new MyThing();
```

This line won't compile, because it doesn't use the true name of the class. The line ought to be

```
sgsware.db.MyThing m = new sgsware.db.MyThing();
```

Programming wouldn't be very much fun if you had to use full class names everywhere. Imagine what a burden it would be, if you had to do so much typing. You would soon find yourself wishing for a way to abbreviate: "I wish I could tell the compiler that every time I type `MyThing`, I really mean `sgsware.db.MyThing`." This wish is granted by Java's import feature.

Importing is a very useful feature with an unusual name. The name comes from earlier object-oriented languages, in which the functionality was a kind of symbolic importation. Now the keyword continues to be used in Java, but the functionality has more to do with abbreviation than with importation. The syntax of `import` is

```
import full.class.Name;
```

You can have as many import statements as you like in a source file. Imports must appear before the class declaration, as shown in the following code. It assumes that you want to use the `Employee` and `Manager` classes of a package called `biz`:

```
1. package sgsware.raytracing;
2. import biz.Employee;
3. import biz.Manager;
4.
5. public class User
6. {
7.     Employee   dagwood;
8.     Manager    dithers;
9.
10.    . . .
11. }
```

Thanks to the imports on lines 2 and 3, you can use abbreviated class names on lines 7 and 8 (and everywhere else in this source file). Without the imports, lines 7 and 8 would have to be

```
7.    biz.Employee   dagwood;
8.    biz.Manager    dithers;
```

Sometimes a source file might need to use many or all of the classes in a large package. It would be cumbersome to type in the names of all those classes, one per import line. In the spirit of supporting abbreviation, you are allowed to use an asterisk (*) in place of a class name. This causes all classes in the package to be imported. So the preceding code could be slightly shortened as follows:

```
1. package sgsware.raytracing;
2. import biz.*;
3.
4. public class User
5. {
6.     Employee   dagwood;
7.     Manager    dithers;
8.
9.     . . .
10. }
```

One last note on importing: A class imports all the other classes in its package automatically. So you never have to do the following:

```
package mypackage;
import mypackage.*;
. . .
```

Now that you understand how packages work, you have a foundation for learning about Java's various access modes. These will be presented in the [next section](#). You also have a basis for understanding Java's core classes. These will be introduced in [Chapter 12, "The Core Java Packages and Classes,"](#) and will be presented throughout the remainder of the book.

Access

Java's access control is based on the idea that certain features of a class should not be usable by other classes. Before you learn the details of access control, let's look at why this idea is sound.

One of the fundamental concepts of object-oriented programming is *data hiding*. This is the practice of making a class's data as inaccessible as possible to other classes. Why would this be beneficial?

Often there are many valid ways to represent information. A temperature might be stored as degrees Kelvin, Celsius, or Fahrenheit. A price might be listed in various currencies. A color might be represented as a name, as red/green/blue levels, as red/yellow/blue levels, or as hue/ saturation/brightness levels. Maintenance considerations might force class code to be rewritten so as to change the internal representation. For example, if an Italian company bought a company in the United States, money representation might be converted from dollars to euros, and temperature representation might be converted from Fahrenheit to Celsius.

Even if data representation does not change, it makes sense to localize the code that knows about representation inside a single class. It is wasteful to make all classes know how data is represented internally, and it creates the risk of bugs (if the other classes misinterpret the internal representation).

Imagine a class called `Thermometer`, which somehow reads a physical thermometer device. A very clean design is to give the class a `getTempCelsius()` method. The method name leaves no room for confusion as to the units of the return value. There could also be `getTempFahrenheit()` and `getTempKelvin()` methods, so that nobody ever has to look up the conversion formulas. Moreover, nobody ever needs to know how temperature is represented within the class. It might be Fahrenheit, Celsius, or Kelvin. It might change from one rev of the class software to another. The benefit to those of us who use the `Thermometer` class is enormous: We never have to worry about the internal representation.

The general principle of data hiding is that an object's data should never be accessed directly from outside the object. Instead, the object's class should provide methods for reading and setting the data. These methods are officially called *accessors* and *mutators*, but they are often called by their nicknames: *getters* and *setters*. An accessor/getter has an empty argument list and returns a data value. A mutator/setter has a void return type and a single argument. By common convention, the name of an accessor method begins with `get`, followed by the property to be retrieved. The name of a mutator begins with `set`, followed by the property to be modified.

To support this data-hiding approach, object-oriented languages provide facilities to let you restrict access to a class's data and methods. In Java, this is done with *access modifier* keywords. Java has three access modifiers:

- `public`
- `private`
- `protected`

These keywords appear before the declarations of the data or methods they apply to. The `public` modifier may also appear before a class definition. Before we define what the various access modes mean, let's look at an example to clarify the syntax:

```
public class AccessExample
{
    public int      x;
    private double  d;
    protected static float f;
    char           c;

    public int getX()
    {
        return x;
    }

    private void printC()
    {
        System.out.println("c = " + c);
    }

    protected void setD(double newD)
    {
        d = newD;
    }

    void bumpX()
    {
        x++;
    }
}
```

Access modifiers cannot apply to data defined within a method. (Since such data ceases to exist after the method returns, we don't need to think about which outside classes may use it.) Notice the declaration of `f`, which is both protected and static. Access modifiers can be freely combined with non-access modifiers such as *static*. Modifier order is unrestricted, so you could equivalently say `static protected float f`; For the sake of readability, it's good practice to align the first character of your variable names on a tab stop, as the preceding example shows.

Java has three access modifier keywords, but four access modes. The fourth access mode is what you get if you don't specify `public`, `private`, or `protected`. This fourth mode is called *default*, although you might also sometimes see it called *package* or *friendly*. In the preceding example, variable `c` and method `bumpX()` both have default access.

Now let's look at what the different access modes do.

Public Access

Public access is completely unrestricted. Classes, data, and methods can be designated public. A public class can be used by any other class. Public data can be read and written by any code (violating the spirit of data hiding). Public methods can be called from any code.

When you run an application, the Java Virtual Machine creates a class loader, which loads your application class. The class loader is itself a class, and your application class must be public so that the loader can load it.

Private Access

The most restrictive access mode is private. Data and methods can be designated private, but not classes. (There is a kind of class called an inner class that can be private or protected, but inner classes are beyond the scope of this book.) A private variable can be written or read only by an instance of the class that defines the variable. A private method can be called only by an instance of the class that defines the variable.

Private access may not be quite as private as you expect. Let's look at an example:

```
1. public class Employee
2. {
3.     private float salary;
4.
5.     public float getSalary()
6.     {
7.         return salary;
8.     }
9.
10.    public void setSalary(float newSalary)
11.    {
12.        salary = newSalary;
13.    }
14.
15.    public boolean earnsMoreThan(Employee other)
16.    {
17.        if (salary > other.salary)
18.            return true;
19.        else
20.            return false;
21.    }
22.
23.    . . .
24. }
```

On line 3, `salary` is declared private. This makes sense, because one's salary should be kept private. The `getSalary()` and `setSalary()` methods are in the spirit of data hiding. Public methods get and set private data, and there are no surprises. But look at line 17, where the code compares the current employee object's salary to the salary of a different employee. Any object that executes line 17 reads a private variable of a different object.

That's just how private access works. Any instance of `Employee` can read and write not just its own private data, but the private data of any instance. Similarly, any instance of `Employee` can call any private method on any instance of `Employee`.

Default Access

Default access is all about packages. Fortunately, you have just learned all about packages, so you're in great shape to learn about default access.

Default access does not correspond to a modifier keyword. Instead, it's the access mode you get when you don't mark a class, variable, or method as public, private, or protected. A default-access class can be used by any instance of any class that's in the same package. A default-access variable can be read or written only by an instance of a class in the same package as the class that owns the default-access variable. A default-access method can be called only by an instance of a class in the same package as the class that owns the default-access variable. In other words, anything with default access can be used by anything in the same package, and it cannot be used from outside the package.

Default access is useful in this common situation: You sell a package of classes that solve a particular problem. A few of the classes are for direct use by your customers. These are public and thoroughly documented (you'll see how this is done in [Chapter 11](#)). The rest of the classes in the package perform purely secondary roles. They are never used directly by your customers, and are used only by the public classes to help in their internal workings.

There is no need for your customers to know about these secondary classes. In fact, everybody is better off if nobody but you knows about them. This is true not just for classes, but for data and methods as well. Some classes, data, and methods are part of your package's publicly visible interface, while others are nobody's business but your own.

We can draw a parallel between private features in a class and default-access features in a package. In a class, private data and methods are only for the internal working of the class. In a package, default-access classes, data, and methods are only for the internal working of the package.

You have already seen packages with default-access features, although you may not have realized it. When a JVM is executed, it builds a package called the *unnamed package*. This consists of all classes in the current working directory that do not explicitly contain package declarations. Consider the example classes you used in the [previous chapter](#): `Employee` and its subclasses `Worker`, `Manager`, and so on. Those classes didn't use packages, so the obvious way to proceed would be to put them all in the same directory, and to compile with a command like `javac *.java`. Eventually, all the class files would exist in the current working directory. Assuming one of those classes had a `main()` method, a JVM could run that application. Then all the classes

would be considered to be in the no-name package. None of the data or methods in those classes had access modifiers (since access modifiers weren't introduced until this chapter), so they got default access invisibly. And everything could work smoothly. Any instance of any class could use any data, and call any method, of any instance of any class.

The no-name package allows your code organization to evolve as your understanding becomes more sophisticated. At first, you don't know about packages or access. Related source files live together in a directory, along with the corresponding class files. Everything can access everything. Later on, you learn about packages and access. You have several reasonable choices for organizing your source code. All source files can be together in the same directory, or the source can be organized into subdirectories that reflect the package structure, or you can use some other scheme. No matter how you organize your source code, your class files are organized automatically (by the compiler, when you use the `-d` option) into directories that reflect the package structure.

If you are developing code that involves more than just a few classes, it's a good idea to use packages. For smaller projects, it's fine to use a single directory and take advantage of the no-name package. In this book, the code examples are as simple as possible. Example classes have package declarations only when using a package is relevant to the topic of the example.

Protected Access

Protected access is default access plus a little bit more. Only data and methods may be protected; classes may not. (Actually, inner classes may be protected, but they are beyond the scope of this book.)

Protected access is useful in a certain interesting situation. You already saw that default access comes into play when you are sharing a package of interrelated classes, some of which will be directly used by your customers. Protected access comes into play when you are sharing a package of interrelated classes, some of which will be *subclassed* by your customers.

It makes sense for your customers to leave your package intact. You certainly don't want dozens of different evolutions of your package out there in the world, one evolution per customer. It is cleaner for everyone if the various subclasses created by your various customers are in separate packages. But the subclasses might want access to non-public data or methods of their superclasses. Even if those desirable variables and methods had default access, they still wouldn't be useful because they would be in a different package. So protected access grants access to subclasses of the class that owns the protected features, even if the subclasses live in different packages.

A protected method may be overridden in any subclass of the class that owns the method, even if the subclass is in a different package. A protected method may be called by any instance of any subclass of the class that owns the method, even if the subclass is in a different package.

Protected data is more complicated than protected methods. If a variable is protected, it is *not* accessible by just any instance of any other-package subclass. It can be accessed only by the instance of the other-package subclass that owns the data.

Let's look at some simple examples. Here's a superclass:

```
package mystuff;
public class Fish
{
    protected float weight;
}
```

Here's a subclass *in a different package* that makes appropriate use of the protected variable:

```
import mystuff.Fish;

package yourstuff; // Different package!
public class Tuna extends Fish
{
    void printWeight()
    {
        System.out.println("I weigh: " + weight);
    }
}
```

The code is legal because any instance of `Tuna` that executes `printWeight()` is accessing its own version of the protected variable `weight`.

Now here's an example that won't compile, to show you what protected access does *not* mean:

```
import mystuff.Fish;

package yourstuff; // Different package!
public class Tuna extends Fish
{
    void printSomeonesWeight(Fish someone)
    {
        System.out.println("Someone weighs: " +
                           someone.weight);
    }
}
```

This code is illegal, because protected access doesn't mean that any `Tuna` may access any other `Tuna`'s protected data. Protected access is different from private access in this regard.

Bear in mind that this restrictive meaning of "protected" only matters in a different-package subclass. The following code is perfectly legal:

```
package mystuff; // Same package as superclass
public class Snapper extends Fish
{
    void printSomeonesWeight(Fish someone)
    {
        System.out.println("Someone weighs: " +
                           someone.weight);
    }
}
```

Here the situation is different, because the subclass and the superclass are in the same package. Remember that protected access is default access plus a little more. Even if `weight` were default instead of protected, it could be accessed by any class in the same package, independent of any superclass-subclass relationships.

Access and Overriding

Java has a rule that seems strange at first glance: When you override a method, the subclass's version may not have a more restrictive access mode than the superclass's version. This is shown in [Table 9.1](#):

Table 9.1: Legal Access Modes for Overriding Methods

Superclass Version Access	Subclass Version Access
Public	public
Protected	public, protected
Default	public, protected, default
Private	public, protected, default, private

This rule seems arbitrarily restrictive, but on closer inspection, it is absolutely necessary. The [previous chapter](#) discussed polymorphism, and you saw what a powerful tool it can be. It turns out that polymorphism is only possible if you have the access/overriding rule. Let's see why this is.

[Chapter 8](#) presented the `Employee` class, which had a `printCheck()` method. `Employee` had a subclass called `Manager`, which had a subclass called `Officer`. The `Officer` class overrode `printCheck()`. You saw that you could have an array, typed as `Employee[]`, that contained references to objects of a variety of classes. Either `Employee` itself or any of its subclasses were allowed, as shown in [Figure 9.6](#).

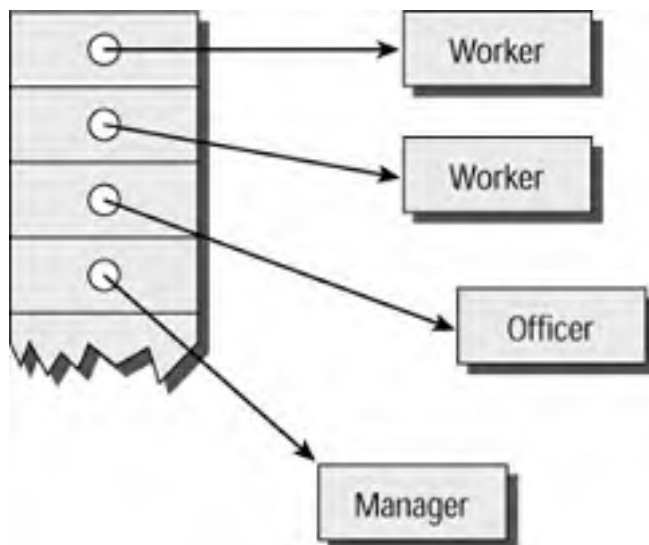


Figure 9.6: Polymorphism revisited

You could then traverse the array as follows:

```
for (int i=0; i<employees.length; i++)
    employees[i].printCheck();
```

Each object would use its own class's version of the `printCheck()` method. This is especially useful if some subclasses override the method.

The clean polymorphic system breaks down if a subclass is allowed to override a method so that the method's access becomes more restricted. To see why this is so, let's assume that the check-printing loop is in some method in a class called `Paymaster`. Let's create a new subclass of `Employee`, called `PartTimer`:

```
class PartTimer extends Employee
{
    private void printCheck()
    {
        // Whatever
    }
}
```

This class won't compile. That's good. If the class were allowed to override `printCheck()` as shown (making its access more restricted, thus violating the rule), it would not be possible for an instance of `Paymaster` to call `printCheck()` on an instance of `PartTimer`. A private method can be called only by an instance of the owning class, so a `PartTimer`'s `printCheck()` could be called only by an instance of `PartTimer`.

Consider what would happen in the absence of this rule. The polymorphic loop in `Paymaster` would work its way through the array, calling `printCheck()` on workers, managers, and officers. Eventually, it would need to make an illegal call to `printCheck()` on a `PartTimer`. This situation must be avoided, and the designers of Java had several options for preventing it. The no-restrictive-overriding rule is sometimes an inconvenience, but actually it is an excellent solution because it gives priority to clean polymorphism.

Team LIB

PREVIOUS NEXT

Final and Abstract

This section will look at two more modifiers: `final` and `abstract`. They aren't access modifiers, but this is still a good place to present them.

Final

Classes, methods, and data may be designated final. A final class may not be subclassed. A final method may not be overridden. A final variable may not be modified after it is initialized.

Final data is useful for providing constants. For example, you might have a `Zebra` class that provides the zebra's weight in pounds or kilograms:

```
class Zebra extends Mammal
{
    private double weightKg;

    public double getWeightKg()
    {
        return weightKg;
    }

    public double getWeightLbs()
    {
        return weightKg * 2.2;
    }
}
```

This class uses appropriate data hiding. A zebra's weight is stored in kilos (the variable name leaves no doubt there), but users of the class never need to know that. Let's assume that eventually the class will have many methods that convert back and forth between kilos and pounds. There will be a lot of multiplying and dividing by 2.2. The standard approach to this situation is to declare a constant:

```
class Zebra extends Mammal
{
    static private final double KGS_TO_LBS = 2.2;

    private double weightKg;

    public double getWeightKg()
    {
        return weightKg;
    }

    public double getWeightLbs()
    {
        return weightKg * KGS_TO_LBS;
    }
}
```

The constant is called `KGS_TO_LBS`. It is static because its value is always going to be the same for all instances of the class, so there is no benefit in giving each instance its own non-static copy. It is private because it is only for use inside the class. It is final because its value should never change under any circumstances. Constants require a little extra typing, but they are well worth the effort for three reasons:

- They explain what they do. Someone reading the code, especially someone who doesn't recognize 2.2 as the kilogram-to-pounds conversion factor, will instantly understand the intention of a constant named `KGS_TO_LBS`.
- They eliminate the need to look up or memorize conversion factors and similar values.
- They provide protection against typos.

The third point requires an example. Suppose you aren't using constants, and it's late at night, and you're tired. Somewhere in the `Zebra` source code, which is now thousands of lines in length, your finger slips and you accidentally type `3.3` instead of `2.2`. It could take a long time for the error to manifest itself, and when it does, you will have to sort through thousands of lines of code to find the problem.

On the other hand, suppose you are committed to using the constant. It is still late at night, and your finger slips, and you accidentally multiply by `KGS_TO_LBX` instead of `KGS_TO_LBS`. The next time you compile your code, the compiler will complain that variable `KGS_TO_LBX` does not exist. When you use constants, the compiler finds your typos for you.

Abstract

Classes and methods may be designated abstract; data may not.

An abstract method has no method body. All the code from the opening curly bracket through the closing curly bracket is gone, replaced with a single semicolon (;). Here is an example of an abstract method:

```
abstract protected double getAverage(double[] values);
```

The `abstract` keyword may be combined with the `public` and `protected` access modifiers. Here you see a method that is both abstract and protected. There is nothing unusual about the declaration part of the method. It only gets strange after the parenthesis that closes the argument list. Where you would expect to find the method body, there is only a semicolon.

When a class has an abstract method, that method's implementation will be found in the class's subclasses. In a moment you'll see an example, but first let's cover a few rules governing abstract classes and methods.

An abstract class may not be instantiated. That is, you are not allowed to call any constructor of any abstract class. Also, if a class contains any abstract methods, the class itself must be abstract. You might say that an abstract class is one that is incomplete: It lacks one or more method implementations.

Suppose you want to create several classes, all of which share some functionality and model similar real-world things. This strongly indicates that the classes should extend a common superclass, which should contain the shared functionality. Every subclass will inherit the common methods, so this is a good object-oriented design. It would not be unusual at this point to realize that there is some functionality that every subclass must have, but that every subclass should do in its own unique way.

For example, you might be writing classes to draw charts. (We won't cover graphics programming until [Chapter 14, "Painting."](#) For now, the important point is the structure, not the content, of the code.) You might decide to create a `Chart` superclass with subclasses, as shown in [Figure 9.7](#).

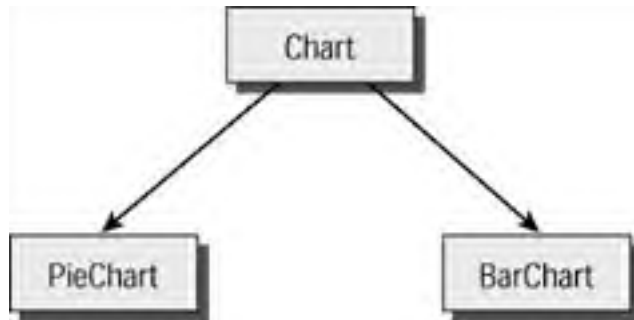


Figure 9.7: Chart class and subclasses

The class will have an array of floats, called `values`, whose values are the values to be charted. The class will also need an array of colors, called `colors`, since the color scheme should be flexible according to the user's taste. Java actually provides you with a class called `Color`. Again, you won't see this class in detail until [Chapter 14](#). For now, you only need to know that the class exists. Its package is `java.awt`, so the code examples that follow all import `java.awt.Color`.

The most important superclass method will be `display()`, whose argument is an array of float values to be charted. An auxiliary method will be `setColorScheme()`, whose argument is an array of colors. Another auxiliary method will be `useColor()`, which has an `int` argument. The argument is an index into the color scheme array. Anything subsequently drawn on the screen, until the next call to `useColor()`, will appear in the specified color.

So far the superclass looks like this:

```
package graphics;
import java.awt.Color;

public class Chart
{
    private float[] values; // Chart these values
    private Color[] colors;

    public void setValues(float[] vals)
    {
        values = vals;
    }

    public void setColorScheme(Color[] newColors)
    {
        colors = newColors;
        display(values);
    }

    private void useColor(int colorIndex)
    {
        // Never mind how this works.
        // You'll see in chapter 14.
    }

    public void display(float[] values)
    {
        for (int i=0; i<values.length; i++)
        {
            useColor(i);
            // ??? Now what ??
        }
    }
}
```

The `useColor()` method is private, since it is for use inside this class only. The other methods are public, since any user might want to call them. The problem is the "???" line in `display()`. It's obvious that for each value to be charted, you should set the appropriate color and then draw a region. You know how to set the color. Ignoring for the moment that you won't learn how to draw on the screen until later in the book, we have a deeper problem. A bar chart and a pie chart draw value regions in different ways. The object-oriented approach tells us that the individual subclasses should encapsulate the knowledge of how to draw appropriately.

Let's convert `Chart` to an abstract class:

```
package graphics;

public abstract class Chart
{
    private float[] values; // Chart these values
    private Color[] colors;

    public void setValues(float[] vals)
    {
        values = vals;
    }

    public void setColorScheme(Color[] newColors)
    {
        colors = newColors;
        display(values);
    }

    private void useColor(int colorIndex)
    {
        // Never mind how this works.
        // You'll see in chapter 15.
    }

    public void display(float[] values)
    {
        for (int i=0; i<values.length; i++)
        {
            useColor(i);
            paintRegion(i, values[i]);
        }
    }

    protected abstract void paintRegion(int n, float value);
}
```

You have added an abstract method: `paintRegion()`. Since the class now contains an abstract method, the class itself must be abstract. Any subclass that doesn't want to be abstract will have to provide an implementation of `paintRegion()`. Since only a non-abstract class can be constructed, the `display()` method in this superclass can trust that it can safely call `paintRegion()`. The true class of the executing object will never be `Chart`. It will be `BarChart`, or `PieChart`, or perhaps some other class to be written in the future. (In the last case, the new class might not be in the same package as the superclass. That's why `paintRegion()` is protected.)

The non-abstract subclasses won't look very interesting, because all their functionality is graphical. Graphical code won't make any sense to you for another few chapters, so here you're just going to see the skeletons of the classes. Here is `PieChart`:

```
package graphics;

public class PieChart extends Chart
{
    protected void paint region (int n, float value)
    {
        // Details not shown. Paint a pie wedge.
    }
}
```

And here is `BarChart`:

```
package graphics;

public class BarChart extends Chart
{
    protected void paint region (int n, float value)
    {
        // Details not shown. Paint a bar.
    }
}
```

Abstract superclasses provide an elegant structure for partitioning shared and unique functionality.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Suppose package `superpack` contains subpackage `subpack`. Suppose a source file contains the following line:

```
import superpack.*;
```


Will this line import classes in `subpack`? Write code to support your answer.
2. Create a class that illegally tries to read a private variable of another class. What is the point of this exercise?
3. Create a class that illegally tries to call a default-access method of another class.
4. Create a class that illegally tries to write a protected variable of another class.
5. True or false: If a class has at least one abstract method, the class must be abstract. Write code to support your answer.
6. True or false: If a class is abstract, it must have at least one abstract method. Write code to support your answer.
7. Write an application that tries to construct an instance of an abstract class. Can you compile the application? Can you execute it?

Chapter 10: Interfaces

The [previous chapter](#) showed you how an abstract class is a class where something is missing. This chapter will present interfaces. An interface is not actually a class, but it's like a class where nearly everything is missing.

A List of Method Declarations

An interface is mostly a list of public method declarations. The source code for an interface is similar to the source for a class in several ways. In particular, an interface definition goes in its own source file, and the source file name should be the interface name, plus `.java` at the end. When the code is compiled, the output file name is the interface name plus `.class`.

Here is an example of an interface. It should appear in source file `Talker.java`, and compilation will produce `Talker.class`:

```
package nature;
public interface Talker
{
    void say(String sayThis);
    void repeat(String repeatThis, int nTimes);
}
```

This interface is in a package called `nature`. Like a class, an interface can belong to a package. It is designated public, so it can be used by any code anywhere. If it were not public, it could be used with the `nature` package only. Interfaces cannot be private or protected.

Before we proceed, it's time to say a little about the `String` class, which appears in the argument list of both methods (and as an array in the argument list of every `main()` method). This is one of many useful utility classes that come with Java. You'll learn about them in [Chapter 12, "The Core Java Packages and Classes"](#). For now, be aware that an instance of the `String` class encapsulates a "run" or "string" of text. The data and methods of `String` won't be used in this chapter.

The list of method declarations appears between the curly brackets that follow the interface name. These declarations are much like abstract method declarations. The return types, method names, and argument lists are present, but the method body is absent, replaced by a semicolon. Unlike abstract methods, the methods declared in an interface are all public. You can declare them as public explicitly if you like, but you may not declare them private or protected. Omitting the access modifier results in public, rather than default, access.

Any class can declare that it implements any interface. This declaration occurs in the class's definition file. The class name is followed by the keyword `implements`, followed by the interface name. For example:

```
package nature;
class Parrot extends Bird implements Talker
{
    . . .
}
```

When a class declares that it implements an interface, the class is saying that it contains an implementation for each of the methods in the interface. (If this is not the case, the class will not compile.) So if the `Parrot` class compiles, you know that it contains a method called `say()` and another method called `repeat()`, with argument lists as specified in the interface.

A class is allowed to implement multiple interfaces. To do this, just provide a comma-separated list of interfaces after the `implements` keyword. So if `Flyer` and `BugEater` are interfaces of the `nature` package, you could have the following class:

```
package nature;
class Mynah extends Bird
    implements Talker, Flyer, BugEater
{
    . . .
}
```

This class would have to provide implementations for the methods of all three interfaces.

Using Interfaces

To see why interfaces are useful, consider the class inheritance hierarchy shown in [Figure 10.1](#).

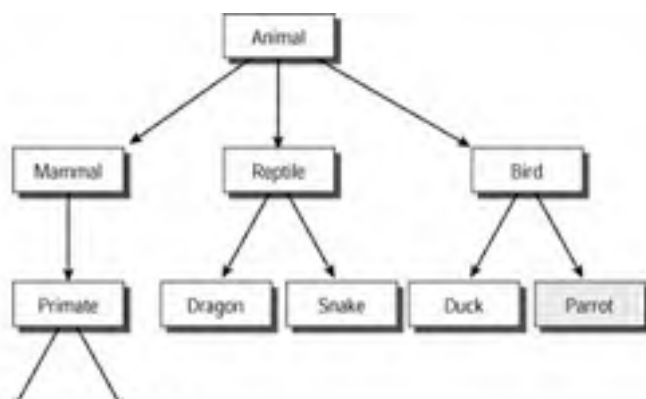




Figure 10.1: Animal kingdom class inheritance

Figure 10.1 would be a very natural way to organize your work if you were creating an extensive library of classes to model the behavior of different kinds of animals. You can imagine behavior such as live birth, cold-bloodedness, and flight being implemented by methods in the `Mammal`, `Reptile`, and `Bird` classes, respectively, and inherited by their respective subclasses. However, there is some behavior that does not fit into the inheritance model.

The three shaded classes in Figure 10.1 share somewhat related behavior. All three species are capable of speech, although the nature of that speech varies greatly from species to species. We humans speak out loud, and we understand what others say. Gorillas can't speak out loud because they don't have the right kind of vocal cords, but they can be taught to use and respond to sign language. (See www.gorilla.org for more information.) Parrots can speak out loud (and do so long after the charm has worn thin), but they do it without comprehension.

A good computer model of the animal kingdom should include speech, so it would make sense to give each of the `Human`, `Gorilla`, and `Parrot` classes its own version of the `say()` and `repeat()` methods described in the previous section. But in that case, the three classes can declare that they implement the interface. For example, `Gorilla` might look like this:

```
package nature;
public class Gorilla extends Primate implements Talker
{
    public void say(String sayThis)
    {
        // Complicated code to produce sign language.
    }

    public void repeat(String repeatThis, int nTimes)
    {
        for (int i=0; i<nTimes; i++)
            say(repeatThis);
    }
    . . .
}
```

So far we have not mentioned any benefit associated with declaring that a class implements an interface. To understand the benefit, let's revisit the issue of objects and references.

Objects and References

You have already seen that a reference is something that uniquely identifies an object. In Java, you don't have variables that are objects. Instead, you have variables that are *references* to objects. In Chapter 8, "Inheritance," you saw that the type of a reference can be different from the class of the object it refers to. This point is important enough that we make a distinction between the *type* of a reference and the *class* of an object. A reference's type is what appears in the declaration of the reference variable; an object's class is the class of the constructor that was invoked when the object was created.

You have already seen that when a reference points to an object, the type of the reference can be exactly the class of the object, or it can be any superclass of the class of the object. Interfaces provide an even wider range of reference types, because reference variable types can be interfaces as well as classes. An interface-type reference can legally point to an object if that object's class implements the interface. So in our ongoing example, the following code would be perfectly legal:

```
Parrot polly = new Parrot();
Talker aTalker = polly;
```

Or even:

```
Talker aTalker = new Parrot();
```

With an interface-type reference, you can call only the methods of the interface. This may seem limiting, but the benefit is huge. Consider the following method:

```
singHappyBirthday(Talker t, String forWhom)
{
    t.repeat("Happy birthday to you.", 2);
    t.say("Happy birthday, dear");
    t.say(forWhom);
    t.say("Happy birthday to you.");
}
```

This method has a talker recite the song, no matter what the class of the talker. A human will sing, a gorilla will sign, a parrot will squawk. This is another example of polymorphism: A single method name (`say`, and also `repeat`) appears in many different forms.

instanceof

Java has a keyword, `instanceof`, that tests the relationship between an object and a reference type. The syntax is

```
<reference> instanceof <type>
```

The reference can be any reference. The type can be the name of any class or interface. The value of an `instanceof` expression is boolean. It is true if a reference of the given type legally can point to the object pointed to by the given reference. For example, this code will print out the message:

```
Duck daffy = new Duck();
if (daffy instanceof Bird)
{
    System.out.println("Yes, Daffy is a bird.");
}
```

So will the following:

```
Bird daffy = new Duck();
if (daffy instanceof Bird)
{
    System.out.println("Yes, Daffy is a bird.");
}
```

And so will the following:

```
Object daffy = new Duck();
if (daffy instanceof Bird)
{
    System.out.println("Yes, Daffy is a bird.");
}
```

The `instanceof` keyword doesn't care about the type of the variable (that is, the word that comes before `instanceof`). What matters is the class of the object to which the variable points, and in all three examples the class is `Duck`. When the second argument of `instanceof` is a class, as in this example, the value is true if the object's class is the same as, or a subclass of, the second argument. Here is an example of `instanceof` where the second argument is an interface:

```
Duck daffy = new Duck();
if (daffy instanceof Talker)
{
    System.out.println("Yes, Daffy is a bird.");
}
```

When the second argument is an interface, the value of an `instanceof` expression is true if the object's class implements the interface. Here, the object's class is `Duck`, which does not implement `Talker`, so the value is false. In the following code, the value is true:

```
Gorilla ndume = new Gorilla();
if (ndume instanceof Talker)
{
    System.out.println("Yes, Ndume can talk.");
}
```

Data in Interfaces

An interface is allowed to contain data, provided the data is public, final, and static. This provides an easy way to define constant data.

In [Chapter 9, "Packages and Access,"](#) you looked at static data within a class with the following example:

```
class Zebra extends Mammal
{
    static private final double KGS_TO_LBS = 2.2;
    . . .
}
```

The variable `KGS_TO_LBS` can be used anywhere within the `Zebra` class. If other classes in the same package want to use the constant, you can declare `KGS_TO_LBS` to have default access (or even protected access, in which case the other-package subclasses can also use it). The other classes can refer to the constant as `Zebra.KGS_TO_LBS`. Sometimes this is fine, but in our example it seems to imply that converting from kilograms to pounds has something to do with zebras. If a constant is more properly associated with a package in general, rather than with any individual class, generally it is better to put it in an interface.

For example, you might be creating a package of classes that model the physics of various mutually interacting heavenly bodies. Your package would be called `astro`, and the classes would be `Planet`, `Star`, `BlackHole`, `Comet`, and so on. (Their official names would be `astro.Planet`, `astro.Star`, `astro.BlackHole`, and `astro.Comet`.) The classes probably would all need to use certain fundamental constants, such as the speed of light and the mass of a proton. Let's look at the various options for implementing these.

First, you can avoid the use of constants altogether. Wherever you need the speed of light, use `3.0e8`; wherever you need the mass of a proton, use `1.67e-27`. As you saw in [Chapter 9](#), this approach is risky. If you type a wrong digit, you'll introduce a bug that can be very hard to find. Moreover, readers of your code might not recognize the significance of `3.0e8` or `1.67e-27` (would you?), so they would not understand the formulas you were implementing.

The next step is to put constants in one of your classes. You might pick `Star`, arbitrarily, and insert the following lines:

```
final static double LIGHT_SPEED = 3.0e8;
final static double PROTON_MASS = 1.67e-27;
```

By convention, constants are in all capital letters, with words separated by underscores. Recall that with constants, a typing error results in a variable name that the compiler will not recognize, so you recruit the compiler to help you find typos.

Before we go further, notice that the constant names can be improved on. As they stand, they are truthful but not entirely helpful. `1.66e-27 whats?` `3.0e8 whats per what?` For optimum clarity, it's best to put the units in the constant names:

```
final static double LIGHT_SPEED_M_PER_SEC = 3.0e8;
final static double PROTON_MASS_KG = 1.67e-27;
```

That takes a little more typing, but now nobody will ever think the speed of light is expressed in miles per second, or proton mass in micrograms. Within the `Star` class, you can refer to the constants by name. Elsewhere in the `astro` package, you can refer to them as `Star.LIGHT_SPEED_M_PER_SEC` and `Star.PROTON_MASS_KG`.

This is certainly better than typing literal constants, but it implies that the constants are somehow naturally associated with the class they appear in. You can go one step further by creating an interface for your constants:

```
package astro;

interface AstroConstants
{
    final static double LIGHT_SPEED_M_PER_SEC = 3.0e8;
    final static double PROTON_MASS_KG = 1.67e-27;
}
```

You can also put method declarations in the interface code, but you don't have to. An interface can declare any number of methods, including zero. Now classes in the `astro` package can refer to `AstroConstants.LIGHT_SPEED_M_PER_SEC` and `AstroConstants.PROTON_MASS_KG`. You don't have to pick a class arbitrarily to put your universal constants in.

You can go one step further, because of the following rule: A class that implements an interface can use the constants of that interface by name, without prefixing the interface name. So your `BlackHole` class could use the following declaration:

```
package astro;

class BlackHole implements AstroConstants
{
    . . .
}
```

You don't have to do any work to ensure that `BlackHole` implements all the methods of the interface, because there are no methods in the interface! And now, anywhere within the `BlackHole` code, and within the code of any other class that implements `AstroConstants`, you can refer simply to `LIGHT_SPEED_M_PER_SEC` and `PROTON_MASS_KG`.

Warning Beware of a subtlety concerning interfaces. All methods and constants in an interface are public. You can use the `public` keyword explicitly for clarity, but if you omit it, the interface's features are still public. They do not have default access, which is what you get if you omit an access modifier in the source code for a class. In an earlier example, when you put the constants in class `Planet`, you didn't use an access modifier. So the constants had default access and could be used anywhere within the `astro` package, but nowhere else. When you moved the constants to the interface, they were forced to be public. Hence, they were accessible from any code regardless of package.

Extending Interfaces

An interface is allowed to extend another interface. The syntax is

```
interface <interface_name> extends <parent_interface>
{
    // Declarations and data
}
```

The new interface consists of all the methods and data defined in the parent interface. For example, you might define the following:

```
package nature;

interface SingingTalker extends Talker
{
    public void sing(String song);
}
```

This interface consists of the two methods defined in `Talker` (`say()` and `repeat()`), as well as `sing()`. A class that wants to implement this interface must use all three methods.

A class can extend only a single parent class, but an interface can extend multiple parent interfaces. For example, if `InterA`, `InterB`, and `InterC` are all interfaces, the following is legal:

```
interface ManyParents extends InterA, InterB, InterC
{
    public int anotherMethod(double d, char ch);
}
```

This interface consists of all the constants and methods of all three parents, plus the method defined explicitly in the source code.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Suppose an interface declares three methods. And suppose a class declares that it implements the interface, but in fact it only implements two out of the three methods. What happens when you try to compile the class? (The way to answer this question, of course, is to write an interface and a class.)
2. If class A implements an interface, any subclasses of A inherit all the methods specified in the interface. Does this mean that subclasses of A also implement the interface? Write code to discover the answer.

3. Given the following interface:

```
interface InterfaceQ3
{
    void printALine();
}
```

Will the following code compile?

```
class ClassQ3 implements InterfaceQ3
{
    void printALine()
    {
        System.out.println("OK");
    }
}
```

4. Don't worry, the following question requires absolutely no understanding of physics. In fact, it might make you grateful that you chose computer programming instead. Suppose you have the following interface:

```
package physics;
interface PhysicsConstants
{
    public static final double ELECTRON_MASS_KG = 9.11e-31;
    public static final double
        STEFAN_BOLTZMANN_CONSTANT_WATTS_PER_M2 = 5.67e-8;
}
```

What does the following application print out?

```
package physics;

public class Q4 implements PhysicsConstants
{
    public static void main(String[] args)
    {
        System.out.println("The value is " +
            STEFAN_BOLTZMAN_CONSTANT_WATTS_PER_M2);
    }
}
```

Chapter 11: Exceptions

Overview

At several points in this book, you have seen how certain program features alter the usual linear flow of program execution. You saw that loops are like whirlpools, conditional statements are like forks in the road, and method calls are like detours.

Extending this analogy, exceptions are like jumping through hyperspace. With exceptions, you can instantly end up far from where you began, with no prospect of getting back. Of course, jumping through hyperspace is an unusual way to travel. And as you might expect, exceptions are to be used only in unusual programming circumstances.

This chapter will show you how to use exceptions to indicate unusual conditions in the state of your programs. It will take a careful approach to this topic, because exceptions are complicated. It would not be helpful to overwhelm you with information. First you'll look at some of the basic concepts of exceptions. Then you'll learn how exceptions are used in real life.

Exceptions Oversimplified

This chapter will begin with an explanation of why exceptions are important. Then it will look at an extended example that uses exceptions. Please bear in mind that the example code here is intended to demonstrate concepts, not to stand as an example of good programming. Later on, once you understand how exceptions affect program flow, you'll see more realistic code examples.

The Trouble with Error Codes

First, let's look at why Java needs a feature to support unusual program status. Imagine a remote database that stores daily rainfall reports for various weather stations. (A remote database is one where the data is stored on a different computer from the one you are using. The two machines are connected by a network.) Without going into the details of how to get data from a remote database into a Java program, imagine a class with a method that somehow retrieves rainfall numbers. The method might be called

```
float getRainfall(int station, int year,
                 int month, int day)
```

For example, to get the rainfall for station #7 for July 8, 2001, you would call

```
getRainfall(7, 2001, 7, 8);
```

You can imagine the method doing network connection stuff, database login/password stuff, database query stuff, network disconnection stuff, and finally returning a value. The problem is, what happens if something goes wrong with the database? Here are a few things that could go wrong:

- The database computer could be turned off.
- The network cables could break.
- The database could be deleted.
- The database management code could crash.
- The database password could be changed.

All of these possibilities are beyond the control of the programmer who is writing the `getRainfall()` method. They are not bugs. A bug is when you write code that doesn't do what you want it to do. Bugs can be avoided by intelligent design and programming, but no amount of programming forethought on your part can prevent someone from walking up to a remote computer and turning it off.

Your code has no straightforward way to deal with these unusual circumstances. There is no straightforward way to tell the method's caller that the database computer was turned off or the password didn't work. Before the invention of exceptions (which predate Java), the only reasonable option was to designate certain special return values, called error codes, to indicate that something unusual happened.

In this example, you might reserve large return values as error codes. You might decide that 10,000 means the password didn't work, 20,000 means the network was unresponsive, and so on. After all, if it ever rains 10,000 inches in a single day anywhere on Earth, we'll all have more pressing problems than data processing to worry about.

What is wrong with this approach? The problem is that anyone, anywhere, who calls `getRainfall()` has to remember to deal with all the error codes:

```
float rainfall = getRainfall(7, 2001, 7, 8);
if (rainfall < 1000)
{
    // Process normally.
}
else if (rainfall == 1000)
{
    // Deal with password problem.
}
else if (rainfall == 20000)
{
    // Deal with unresponsive net.
}
```

The error-processing code might display a message, or it might be more sophisticated. The password-handling code might try a different password. The network-handling code might retry the query at one-second intervals, or it might page a system administrator. But no matter how the errors are handled, anyone who calls the method has to check all return values to see if an error code was returned. To do this, they need good documentation that describes each code and its meaning. The programming language can do nothing to support error handling, because from the compiler's point of view, an error code is just an ordinary value returned by a method.

Special problems are introduced when the method is revised, if the new revision introduces new error codes. Now all the old documentation is incomplete. It's even worse if the new rev of the method changes the significance of an existing code.

Things can get worse yet. What if you realize that your error codes actually represent legitimate return values? Certainly, there's nowhere on Earth where it rains 10,000 inches in a day. But on other planets, with active atmospheres and extremely long days (Mercury, for example), 10,000 is common.

Less imaginatively, the data might be gathered automatically into the database by electronic rain gauges. If the electronics fail, a gauge could erroneously report a measurement of 10,000. Then, when `getRainfall()` returned the value, the error-handling code might page a system administrator, who would be paid overtime for rushing to the office at 4:30 in the morning. The administrator would spend hours determining that the network was healthy, and would probably be grouchy for the rest of the day.

You have probably realized that rainfall can never be less than zero, so you should have reserved negative error codes, rather than large ones. That would make your method interplanetary, but the other problems would remain. Still, unless you use exceptions, error codes are the only option. By the time you finish this chapter, you should be an enthusiastic user of exceptions.

Throwing Exceptions

In this section, you will meet two new Java keywords: `throw` and `throws`. They look almost identical, but `throw` only appears in executable code, while `throws` only appears in method declarations. You will also meet the `Exception` class. By now, you are aware that Java uses a number of classes that are provided for you by the system. The `Object` class is an obvious example, and you have also seen a little bit of the `String` class. In [Chapter 12, "The Core Java Packages and Classes,"](#) you will learn about more of these classes. They are too numerous to describe in detail, but you will also learn where to find out about provided classes as needed. But in order to learn that, you first have to understand exceptions.

To see exceptions in action, let's change the `getRainfall()` example. For now, let's suppose that only one unusual condition is recognized by the code: a crashed database. To detect this condition, assume you have a boolean method called `databaseOk()` that returns true if the database is healthy and false if it has crashed. If you were to use the error-code approach, you might write the following:

```
1. float getRainfall(int station, int year,
2.                  int month, int day)
3. {
4.     if (databaseOk() == false)
5.         return -1;
6.
7.     // Get & return rainfall from db
8. }
```

You use `-1` as an error code to indicate a crashed database. Now here is the same method, rewritten to use an exception:

```
1. float getRainfall(int station, int year,
2.                  int month, int day) throws Exception
3. {
4.     if (databaseOk() == false)
5.     {
6.         Exception x = new Exception("The db crashed.");
7.         throw x;
8.     }
9.     // Get & return rainfall from db
10. }
```

This changes the code in 3 ways:

- It adds `throws Exception` to the declaration on line 1.
- It creates an instance of the `Exception` class on line 6.
- It throws the exception (whatever that means) on line 7.

The addition of `throws Exception` to the declaration announces that this method now might throw an exception. Any particular call to the method might or might not throw, but any code that calls the method must be prepared to deal with the possibility.

Line 6 is just an ordinary constructor call. Until they are thrown, exceptions are just ordinary objects. There are two commonly used versions of the `Exception` constructor: a no-args version, and the version used here, which takes a string of text as an argument. The text can be retrieved later by calling the exception's `getMessage()` method. Later on, you will see how this is useful when processing exceptions.

Line 7 is the big idea. The `throw` keyword must be followed by an exception. (Strictly speaking, a few other things can follow `throw`, but they are beyond the scope of this book.) When the `throw` statement is executed, the current execution of the current method is abandoned immediately. Execution jumps (as if through hyperspace!) to the appropriate exception-handling code for the particular exception that was thrown. The exception handler uses the `catch` keyword, which is presented in the [next section](#).

Catching Exceptions

It stands to reason that things that are thrown ought to be caught. This is true for balls, Frisbees, kisses, and exceptions.

When you call a method that declares that it throws an exception, the calling code can't just call the method. For example, the following code will not work:

```
float rainfall = getRainfall(7, 2001, 7, 8);
```

This call used to be fine, but now `getRainfall()` throws an exception, which any calling code must be prepared to catch. The call has to look something like this:

```
1. try
2. {
3.     float rainfall = getRainfall(7, 2001, 7, 8);
4.     System.out.println("rainfall was " + rainfall);
5. }
6. catch (Exception x)
7. {
8.     System.out.println("getRainfall() failed.");
9.     System.out.println("Message is: " + x.getMessage());
10. }
11. System.out.println("And life goes on.");
```

The code on lines 2-5 (in the curly brackets immediately after `try`) is called a *try block*. There are two rules to know about try blocks:

- Any code that throws an exception must appear in a try block. (For now. Later you'll learn how to get around this rule.)
- At least one statement in the try block must throw an exception.

The code on lines 7-10 (in the curly brackets immediately after the `catch` line) is a *catch block*. When a statement in a try block is executed and causes an exception to be thrown, the current pass through the try block is abandoned immediately. Execution jumps to the first line of the catch block.

Note the code in parentheses on line 6, after `catch`. It looks like a variable declaration, and indeed it is. When an exception is thrown, the JVM makes the exception object accessible to the catch block. When you declare `Exception x`, you are saying that you want to use the variable name `x` as the name of your reference to the exception. This variable has scope (that is, valid meaning) only within the catch block.

Notice line 9, which makes a method call on `x`. Recall that you can pass a message into the `Exception` constructor. The message is stored in the exception object, and the `getMessage()` call retrieves it. Recall from the [previous section](#) that `getRainfall()` stored a message that said, "The db crashed."

Note If you are at all uncomfortable with the way line 9 adds literal text to a method call, please be patient. All will be explained in the [next chapter](#). For now, just be aware that it works.

What happens if the try block runs in its entirety, with no exception being thrown? In this case, the catch block is ignored. Execution jumps around the catch block, from line 4 to line 11. If this is the case, and if the rainfall value is 1.45 inches, the code will produce the following output:

```
Rainfall was 1.45  
And life goes on.
```

On the other hand, if the call to `getRainfall()` throws an exception, the output will be
`getRainfall()` failed.
Message is: The db crashed.

The Simple Exception Lab animated illustration demonstrates the flow of execution through code, which is almost identical to this example. To run the program, type `java exceptions.SimpleExceptionLab`. You will see the display shown in [Figure 11.1](#).

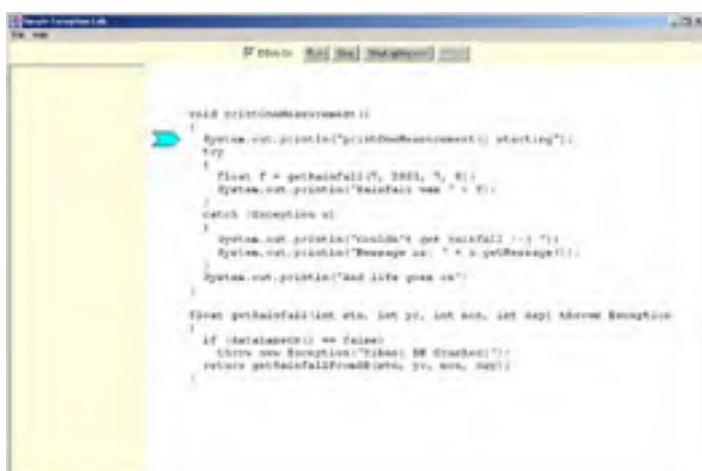


Figure 11.1: Simple Exception Lab

You will see the code for a method that calls another method to retrieve a rainfall measurement from an imaginary remote database. (There isn't really a remote database, and your computer doesn't have to be connected to a network for the animated illustration to work.) You can use the checkbox to control whether the imaginary database is working or not. As [Figure 11.1](#) shows, the DB (database) is initially okay. If you uncheck the checkbox, the second method will throw an exception that will be caught by the first method. The text area to the left of the display will show all output from the `println` statements. Try running the program once with the DB is Ok checkbox checked, to simulate normal execution. [Figure 11.2](#) shows the final state.

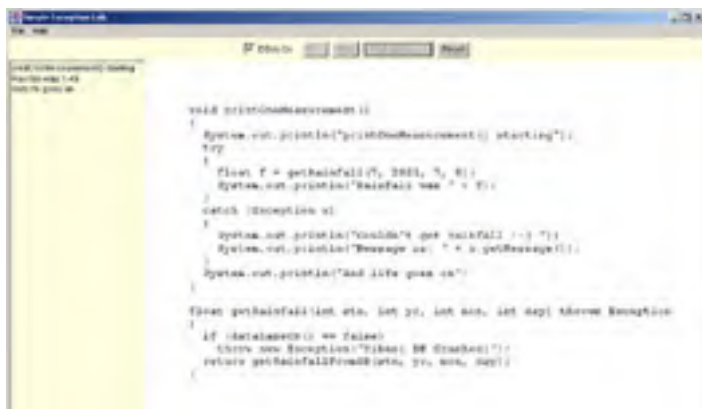




Figure 11.2: Simple Exception Lab: final state with normal execution

Think about what the code would print out if the database wasn't okay. Run it again with the checkbox unchecked to observe the error-handling behavior. Was the output what you expected?

Team LIB

PREVIOUS NEXT

Exceptions in the Real World

So far, the point of this chapter has been to familiarize you with exceptions, and especially with trying, throwing, and catching. Now that you understand these concepts, it is time to tell you that the situation is actually a lot more complicated. There are many kinds of exceptions, each one indicating a different kind of problem, and each one capable of being handled separately.

The remainder of this chapter will show you how to deal with the multitude of real-world exceptions.

Two Families of Exceptions

The `Exception` class has more than 100 subclasses in the core Java packages. (The *core packages* are the ones that you get along with the JVM and the Java compiler. You can think of them as the infrastructure of Java, providing classes that are essential to the operation of the JVM, the compiler, and your own applications.)

The two families are

- Checked exceptions
- Runtime exceptions

The `Exception` class has a subclass called `RuntimeException`. The family of runtime exceptions consists of the `RuntimeException` class and all its subclasses. The family of checked exceptions consists of all other exception classes, including `Exception` itself. (Note that there is no `CheckedException` class.)

Generally, exception classes have long and descriptive names, such as `PrinterIOException` and `ArrayIndexOutOfBoundsException`. Usually, the class name tells you very specifically what went wrong. Let's use these two classes to look at the difference between checked and runtime exceptions.

`PrinterIOException` is a checked exception. It's thrown by methods that interact with a printer. If a printer is jammed, unavailable, or in some other failure state, the method throws `PrinterIOException`. `ArrayIndexOutOfBoundsException` is a runtime exception. As you can guess from the name, it is thrown when an array index is \geq the length of the array, or when the index is negative.

What's the difference between these two situations? It all comes down to who is responsible for creating the problem. In the case of `PrinterIOException`, you can't really say it's anyone's fault. Printers jam up or fail in other ways that are familiar to all owners of printers. That's an environmental hazard. It's unavoidable, like bad weather. On the other hand, with `ArrayIndexOutOfBoundsException`, it's easy to assign blame. The programmer who wrote the line of code that used the illegitimate array index should have done a better job. After all, it would be ridiculous to tell you to turn to [page 1,963](#) in this book... or worse yet, to [page -47](#). Similarly, you shouldn't refer to an array element that doesn't exist.

To generalize from these examples: All checked exceptions represent situations that are unavoidable. All runtime exceptions represent situations that can be avoided by better programming. This implies that your Java programs might sometimes throw checked exceptions, but they should never throw runtime exceptions.

The proper way to deal with checked exceptions is with the try/catch mechanism described earlier in this chapter. The proper way to deal with runtime exceptions is... well, you should never have to deal with them, because your code should never throw them. Of course, code is never perfect the first time you write it. Whenever you write a long piece of code, your first job is getting it to compile. Once you do that, you're only halfway finished. The next step is to make your code run correctly by finding and eliminating bugs. During this phase of development, you are likely to encounter runtime exceptions, and your job is to eliminate them. So your finished, polished, ready-for-market code should never throw runtime exceptions. During development, runtime exceptions are signposts that point to code that needs fixing.

Runtime exceptions should *not* be caught in catch blocks. But how can this be? Earlier in this chapter, you learned that if code might throw an exception, the code has to appear in a try block and the exception has to be caught in a corresponding catch block. Well, that was an oversimplification to avoid giving you too much information all at once. Now that you're half an expert on exceptions, you can learn the whole story.

Code that throws checked exceptions *must* appear in a try block, with the exception caught in a catch block. But this rule *does not apply* to code that throws runtime exceptions. Such code *may* appear in a try/catch structure, but it doesn't have to, and usually it should not. Instead, the code that would throw the runtime exception should be fixed so that it no longer throws.

Runtime Exceptions and Stack Traces

Now you know that you should *not* catch runtime exceptions. But then what happens when one is thrown?

When any kind of exception is thrown, the JVM stores some very useful information in the exception. This information is called the *stack trace*, and often it's all you need to find the source of the problem. The stack trace tells you what line of code threw the exception, as well as the name of the method that contains the line. The stack trace also tells you what line of code called that method, and so on. It goes back and back until you get the line in your `main()` method that called the method that called the method that called the method that owned the line that threw the exception. It's like *This Is the House That Jack Built*, only it's about a Java program instead of a house:

This is the program that you built.

This is the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

This is the line that threw the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

This is the method that owns the line that threw the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

This is the line that calls the method that owns the line that threw the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

This is the method that owns the line that calls the method that owns the line that threw the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

...

This is the `main()` method that owns the line that calls the method that owns the line that calls the method that owns the line... that threw the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

Let's look at a practical example. Suppose you have the following application:

```
1. public class ShowMeATrace
2. {
3.     public static void main(String[] args)
4.     {
5.         int[] cubes = new int[10];
6.         storeCubes(cubes);
7.     }
8.
9.     private static void storeCubes(int[] intArr)
10.    {
11.        for (int i=0; i<=10; i++)
12.            storeOneCube(intArr, i);
13.    }
14.
15.    private static void storeOneCube(int[] ints,
16.                                     int index)
17.    {
18.        ints[index] = index*index*index;
19.    }
20. }
```

The `main()` method creates an array that's passed to `storeCubes()`. The `storeCubes()` method loads each array component with the cube of its index. It does this by calling `storeOneCube()` once for each component. When you run this application, you get the following output:

```
java.lang.ArrayIndexOutOfBoundsException
    at ShowMeATrace.storeOneCube(ShowMeATrace:18)
    at ShowMeATrace.storeCubes(ShowMeATrace:12)
    at ShowMeATrace.main(ShowMeATrace:6)
Exception in thread "main"
```

This output is a stack trace. Reading from top to bottom, you find that an `ArrayIndexOutOfBoundsException` was thrown from line 18 in the `storeOneCube()` method. The offending call to `storeOneCube()` was made on line 12 in `storeCubes()`, which was called from line 6 in `main()`. (By the way, notice that the first line of the trace implies that `ArrayIndexOutOfBoundsException` belongs to the `java.lang` package. The core Java classes belong to a package called `java`, which contains many subpackages. The most important subpackage is `java.lang`, which contains a large number of vital infrastructure classes. You will look at some of these classes in the [next chapter](#).)

So the stack trace tells you to pay attention to lines 18, 12, and 6. Usually your best strategy is to look at lines in the order they appear in the trace. Line 18 seems innocent, as long as `index` is reasonable. But `index` is supplied by the method's caller, so you look at line 12. You see that `index` in `storeOneCube()` corresponds to `i` in `storeCubes()`. The maximum value of `i` is 10, but the array only has 10 components, so the maximum legal index is 9. You have found the problem.

There are two ways to fix the bug. The lazy way would be to change line 11 like this:

```
for (int i=0; i<10; i++)
```

That would solve the problem at hand, but if the array size (in `main()`) ever changes, you will have to remember to change line 11. The safe way, which is better style in all cases, is to use the following for line 11:

```
for (int i=0; i<intArr.length; i++)
```

If you have a program that uses an array, it is very likely that eventually you will create a for loop to do some kind of processing on each array component. If you use the kind of for loop shown here, you will always be sure to process every component while avoiding `ArrayIndexOutOfBoundsException`.

Warning Be aware that some versions of the JVM do not provide stack traces when exceptions are thrown. This usually happens because the JVM performs some kind of optimization that makes it impossible to piece together the stack trace information. When these machines throw an exception, you just get a message that tells you the class of the exception.

Checked Exceptions

In the [previous section](#), you saw that you should not catch runtime exceptions, even though the language allows you to. However, when you call a method that throws a checked exception, you have no choice but to use the try/catch mechanism. If you don't, your code will not compile.

Suppose you have a method, called `printRetAddr()`, that prints your return address on an envelope. Assume you have the kind of printer that can detect whether it is loaded with paper or envelopes. If it is not loaded with envelopes, the method throws `PrinterIOException`, which is a checked exception. If you want to call the method, your code might do the following:

```
void printSomeEnvelopes(int nEnvelopes)
{
    for (int i=0; i<nEnvelopes; i++)
        printRetAddr();
}
```

Simple enough, but it won't compile. Your compiler error will be something like this:

PrinterIOException must be caught or declared to be thrown at line xx, column xxx.

This tells you that you have two options. Your first option is to put the call to `printRetAddr()` inside a try block:

```
void printSomeEnvelopes(int nEnvelopes)
{
    for (int i=0; i<nEnvelopes; i++)
    {
        try
        {
            printRetAddr();
        }
        catch (PrinterIOException piox)
        {
            System.out.println("Please load printer " +
                               "with envelopes.");
        }
    }
}
```

Earlier in this chapter, you saw code that catches `Exception`. Here you see that any subclass of `Exception` may be caught (as long as it really is thrown in the try block; see Exercise 4 at the end of this chapter). The catch block will be executed if the try block causes a `PrinterIOException` to be thrown.

You have a second option. If you don't want to use try/catch, you can simply declare that `printSomeEnvelopes()` throws `PrinterIOException`:

```
void printSomeEnvelopes(int nEnvelopes)
    throws PrinterIOException
{
    for (int i=0; i<nEnvelopes; i++)
        printRetAddr();
}
```

Now any method that calls `printSomeEnvelopes()` must either use try/catch or declare that it too throws `PrinterIOException`.

Multiple Catch Blocks

Typically, code in a try block can throw more than one kind of exception. To illustrate this, let's look at another type of checked exception: `ConnectException`. This is usually thrown by code that attempts to connect to a machine on the network, such as a Web server. If the remote machine does not respond (because it has been turned off, or is undergoing maintenance, or has burned up), the code that detects the lack of response should throw a `ConnectException`. (You will look at network connections in detail in [Chapter 13, "File Input and Output."](#) For now, the point is that now you know about two checked exception types.)

To extend this example, let's make `printSomeEnvelopes()` more responsible. Suppose you have two utility methods at your disposal:

`getNumEnvelopesInStock()` Returns the number of envelopes left, not counting the ones you just printed. This value is retrieved from a remote database.

`setNumEnvelopesInStock()` Updates the number of envelopes left. This value is stored on the remote database.

Both methods throw `ConnectException` if the machine where the remote database resides cannot be contacted. Now `printSomeEnvelopes()` can be written like this:

```
void printSomeEnvelopes(int nEnvelopes)
{
    for (int i=0; i<nEnvelopes; i++)
    {
        try
        {
            printRetAddr();
        }
        catch (PrinterIOException piox)
        {
            System.out.println("Please load printer " +
                               "with envelopes.");
            return;
        }
    }

    try
    {
        int nEnvelopesLeft = getNumEnvelopesInStock();
        nEnvelopesLeft -= nEnvelopes;
        setNumEnvelopesInStock(nEnvelopesLeft);
    }
    catch (ConnectException conx)
    {
```

```
        System.out.println("Couldn't connect.");
    }
}

System.out.println("printSomeEnvelopes() done.");
}
```

The second try block updates the remote database, taking into account the number of envelopes that were just printed. When `printSomeEnvelopes()` is called, there are four possibilities:

The code could run normally, with no exceptions being thrown. Neither catch block is executed. The method prints the "done" message and then returns.

A `PrinterIOException` is thrown from `printRetAddr()`. Execution jumps to the first catch block, which prints the "Please load..." message and then returns. (It returns because no envelopes were used, so the number in the database shouldn't be decremented. If the catch block did not return, the second try block would be executed.)

A `ConnectException` is thrown from `getNumEnvelopesInStock()`. Execution jumps to the second catch block, which prints the "Couldn't connect" message. Then execution continues after the catch block. The "done" message is printed, and then the method returns.

A `ConnectException` is thrown from `setNumEnvelopesInStock()`. Just as in the previous case, execution jumps to the second catch block, which prints the "Couldn't connect" message. Then the "done" message is printed and the method returns.

This code can be simplified. A single try block is allowed to throw multiple exception types, provided there is a catch block for each type. This might require multiple catch blocks for the try block:

```
void printSomeEnvelopes(int nEnvelopes)
{
    try
    {
        for (int i=0; i<nEnvelopes; i++)
            printRetAddr();
        int nEnvelopesLeft = getNumEnvelopesInStock();
        nEnvelopesLeft -= nEnvelopes;
        setNumEnvelopesInStock(nEnvelopesLeft);
    }
    catch (PrinterIOException pioex)
    {
        System.out.println("Please load printer " +
            "with envelopes.");
    }
    catch (ConnectException cx)
    {
        System.out.println("Couldn't connect.");
    }
    System.out.println("printSomeEnvelopes() done.");
}
```

The work has been consolidated into the single try block. There are two catch blocks. If the try block threw five or 50 exception types, there could be five or 50 catch blocks.

When the JVM detects a thrown exception in the try block, it scans the various catch blocks. The current pass through the try block is abandoned, and execution continues in the first catch block that is appropriate to the type of thrown exception. This version of the method behaves exactly like the previous version, but it's easier to read because all the normal execution code appears in the try block, while problems are handled in the various catch blocks. No matter how many catch blocks there are, a single thrown exception is only handled by one catch block. After the catch block runs (and it doesn't contain a `return` statement), execution continues after the last catch block.

Catch Blocks and *instanceof*

The [previous section](#) introduced multiple catch blocks. You learned that execution continues in the first catch block that is appropriate to the type of thrown exception. But what makes a catch block appropriate? You might think that the type declared in parentheses after `catch` must match the class of the exception that was thrown. But this is not the whole story. The whole story involves `instanceof`.

Recall from [Chapter 10, "Interfaces"](#), that the syntax for `instanceof` is

```
<reference> instanceof <type>
```

If the type is a class name, and the reference points to an object whose class is either the type or a subclass of the type, `instanceof` evaluates to `true`.

When the JVM looks for a catch block to handle an exception, it uses `instanceof` to determine whether or not a particular catch block is appropriate. To illustrate, let's blur the `printSomeEnvelopes()` example:

```
void printSomeEnvelopes(int nEnvelopes)
{
    try
    {
        // STUFF
    }
    catch (PrinterIOException pioex)
    {
        // STUFF
    }
    catch (ConnectException cx)
```

```
    {  
        // STUFF  
    }  
    System.out.println("printSomeEnvelopes() succeeded.");  
}
```

If the try block throws, the JVM asks if the exception is an instance of `PrinterIOException`. If so, the first catch block is executed. Otherwise, the next catch block is tested. The JVM asks if the exception is an instance of `ConnectException`. If so, the second catch block is executed. In either case, only the one catch block is executed; the other is ignored. If the executing catch block does not return, execution then proceeds at the first statement following the last catch block.

If no exception is thrown, the try block runs to completion and both catch blocks are skipped.

Sometimes you can take advantage of how the JVM determines the appropriate catch block. In the last revision of our example, the two different kinds of exceptions were handled differently, but this might not always be the case. Suppose you decide that no matter what kind of trouble crops up, `printSomeEnvelopes()` should just print a message that says "Could not print" and then return. If there is no trouble, the method should print "Succeeded."

Both `PrinterIOException` and `ConnectException` are subclasses of a common superclass called `IOException`. So `printSomeEnvelopes()` can be rewritten like this:

```
void printSomeEnvelopes(int nEnvelopes)  
{  
    try  
    {  
        for (int i=0; i<nEnvelopes; i++)  
            printRetAddr();  
        int nEnvelopesLeft = getNumEnvelopesInStock();  
        nEnvelopesLeft -= nEnvelopes;  
        setNumEnvelopesInStock(nEnvelopesLeft);  
        System.out.println("Succeeded");  
    }  
    catch (IOException iox)  
    {  
        System.out.println("Could not print");  
    }  
}
```

Now, any kind of exception that the try block might throw will pass the instance of `IOException` test, so execution will end up in the single catch block.

You can get even more sophisticated. There is a kind of catch block that is informally called a *safety net catch block*. This is not official Java terminology, but it's very commonly used. You might have a try block that throws many subclasses of `IOException`, including `PrinterIOException` and `ConnectException`. Suppose those two types require individual handling, but all other types can be handled the same. You could do the following:

```
try  
{  
    // Lots  
    // and  
    // lots  
    // and  
    // lots  
    // of code that throws  
    // lots  
    // and  
    // lots  
    // and  
    // lots  
    // of subclasses of IOException  
}  
catch (PrinterIOException piox)  
{  
    // Special PrinterIOException handling  
}  
catch (ConnectException cx)  
{  
    // Special ConnectException handling  
}  
catch (IOException iox)  
{  
    // General IOException handling  
}
```

If either `PrinterIOException` or `ConnectException` is thrown, the appropriate specific catch block will be executed. If a different type of `IOException` is thrown, the JVM will first check if the exception is an instance of `PrinterIOException`. It isn't, so next the JVM will check if it is an instance of `ConnectException`. Again, it isn't, so the JVM checks if it is an instance of `IOException`. And it is, because subclasses pass the instance of test, so the last catch block is executed. You can see how the last catch block is a kind of safety net, catching all `IOExceptions` that aren't caught by the two specific catch blocks.

The safety net block could have caught `Exception` instead of `IOException`. The code would have identical behavior, but the safety net is overly general and is considered bad coding style. The exception type caught by a safety net should be the lowest-level subclass that gets the job done. See Exercise 5 at the end of this chapter to find out why.

When you use a safety net, be careful about the order of appearance of your catch blocks. Don't do the following:


```
try
{
    // Something
}
catch (IOException iox)
{
    // General IOException handling
}
catch (PrinterIOException piox)
{
    // Special PrinterIOException handling
}
catch (ConnectException cx)
{
    // Special ConnectException handling
}
```

The second and third catch blocks can never be executed, because both `PrinterIOException` and `ConnectException` pass the instanceof `IOException` test. The compiler will not allow this code. You will get a compiler message that says something like, "Catch is unreachable at line xxx."

The Advanced Exception Lab animated illustration shows exception handling in situations where the try block can throw multiple exception types from any of several lines. To start the program, type `java exceptions.AdvancedExceptionLab`. You will see the display shown in [Figure 11.3](#).



Figure 11.3: Advanced Exception Lab

You get to choose the type of exception that will be thrown. Click on the Choose Type... button and you will see a dialog that lets you choose from four checked types, as shown in [Figure 11.4](#).



Figure 11.4: Choosing an exception type in Advanced Exception Lab

You can click on any of the exception types except the `Exception` superclass. You can also choose which line throws the exception by clicking on the checkbox on the line of your choice on the main screen. The lab lets you choose one of five code configurations (via the File?Configurations menu). In each configuration, a method called `top()` calls a method called `middle()`, which calls a method called `bottom()`. The bottom method has a try block from which an exception is thrown. Different configurations handle the exception differently. [Figure 11.5](#) shows the Spread Around configuration, with an `AWTException` thrown from method `ccc()`.

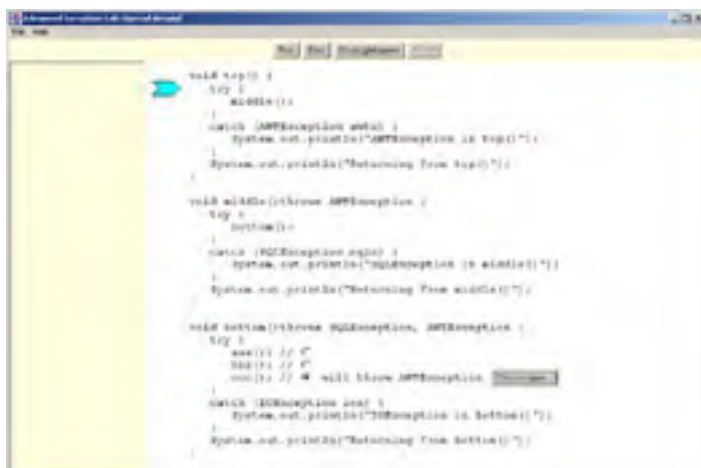


Figure 11.5: Advanced Exception Lab reconfigured

Try all the configurations, and be sure that the exception handling makes sense to you in all cases.

Checked Exceptions and Stack Traces

You have already looked at stack traces in the context of runtime exceptions. Checked exceptions also have stack traces. However, you usually don't see the trace because you have to catch checked exceptions. If you want to see a stack trace from a checked exception, you can call the `printStackTrace()` method:

```
try
{
    getNumEnvelopesInStock();
}
catch (ConnectException cx)
{
    System.out.println("Stress!");
    cx.printStackTrace();
}
```

If an exception is thrown, this code will print the "Stress" message, followed by the exception's stack trace. This is extremely useful during development. However, before you ship your code to paying customers, you might want to delete the `printStackTrace()` calls. Paying customers might not want that much information.

Throwing Checked Exceptions

If you're writing a method that throws exceptions, you have to decide which exception type to throw. You have three options:

- Throw `Exception`, as in the examples in the first half of this chapter.
- Throw a subclass of `Exception` from the core Java classes.
- Throw your own custom subclass.

The first option is not realistic. Your code will work, but throwing `Exception` doesn't tell anybody else who reads your code anything about the nature of the exceptional condition. Also, you may need to call your method in a try block that calls other methods that throw. If your method throws `Exception`, it may be difficult or impossible to write a decent set of catch blocks. This is especially true if the try block calls more than one method that throws `Exception` when it could have thrown a more specific type. Your code is always most robust when you throw exceptions that are as specific as possible.

Your second option is to throw a preexisting exception type chosen from the core Java classes. This is easy when you know how to explore the core Java packages and discover the names and behaviors of the many classes they provide. You will learn how to do this in the [next chapter](#). For now, be aware that it's important to choose the most accurate and informative exception name you can find. Most existing types have very long and informative names.

Unfortunately, a lot of programmers always throw `IOException`, even when the problem has nothing to do with Input/Output. This is a bad habit. The rationale seems to be that, out of all the checked subclasses of `Exception`, `IOException` has the shortest name. Please don't yield to this temptation.

Once you decide on an exception type, you construct and throw just as you saw earlier in this chapter, when you constructed and threw `Exception` (which, as you now understand, you should never do). All exception subclasses in the core Java packages have two forms of constructors: a no-arguments version, and a version that takes a text message as an argument. It is always better to use the second form. Be sure to compose a message that is both accurate and helpful.

For example, earlier in this chapter you saw code that called a hypothetical method called `getNumEnvelopesInStock()`, which threw `ConnectException`. Put yourself in the shoes of the person who wrote that method. He might have done something like the following, assuming he had a method called `connectionOK()` that returned `true` if the connection to the database server was sound:

```
public int getNumEnvelopesInStock()
    throws ConnectException
{
    if (connectionOK() == false)
        throw new ConnectException();

    // Get & return # of envelopes remaining.
    . . .
}
```

However, it would be more informative to include a message in the exception. You might pass something like the following into the constructor: "Couldn't get # of envelopes from remote db." Then any catch block that caught your exception could call `getMessage()` on it, printing out the result if appropriate.

What should you do if there's no appropriately named exception subclass in the core packages? You have to fall back on your third option, which is to create your own class. To do this, first decide if you should create a checked exception or a runtime exception. In other words, does your exception represent an unavoidable hazard of existence, or is it a programming error that should be fixed? Most often you will create a checked exception. Next, choose a name. The name should end with `Exception`, because that's what other people expect. For example, you might decide that if the `getNumEnvelopesInStock()` method can't connect to its remote database, it should throw a custom exception type. A plausible name would be `RemoteEnvelopeCountException`. The name says that the class is definitely an exception, and that both remote access and the envelope count are involved.

Having chosen a name, next you have to decide on a superclass. A checked exception should extend `Exception`, `IOException`, or some other checked exception type. In general, extend `IOException` or one of its many subclasses if the exceptional condition you want to represent involves input or output. Otherwise, extend `Exception`. In the rare case when you want to create a runtime exception, extend `RuntimeException`. In this example, `RemoteEnvelopeCountException` will be a subclass of `ConnectException`, since the problem stems from an inability to connect to the remote machine that owns the database.

Your custom class does not need any data or methods. It will inherit everything it needs. All you have to do is create constructors. A custom exception should have both constructor versions. Here is the source for `RemoteEnvelopeCountException`, in its entirety:

```
import java.net.*;

class RemoteEnvelopeCountException
    extends ConnectException
{
    RemoteEnvelopeCountException() { }

    RemoteEnvelopeCountException(String s)
    {
        super(s);
    }
}
```

The import line is required because the `ConnectException` superclass lives in the `java.net` package, which is one of the core Java packages that you'll see in the [next chapter](#). The first constructor is a no-arguments constructor that seems to do nothing (but remember the chain of construction from [Chapter 8, "Inheritance"](#)). The second constructor takes a text message, which is passed to the superclass constructor.

Custom exceptions are thrown and caught just like standard types, so you could call `getNumEnvelopesInStock()` like this:

```
try
{
    int n = getNumEnvelopesInStock();
    System.out.println(n + " envelopes remaining in stock.");
}
catch (RemoteEnvelopeCountException recx)
{
    System.out.println("Stress!");
    System.out.println(recx.getMessage());
}
```

Generally, it's better to use an existing exception type if you can find one whose name accurately and helpfully describes the exceptional condition. However, if no such class exists, creating a custom class is good programming style.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. What happens when you run a program that creates an array of ints and then sets the value of an array component whose index is greater than the length of the array?
2. What happens when you run a program that creates an array of ints whose length is less than zero?
3. What happens when you run a program that prints out the result of dividing a non-zero int by zero?
4. Write a program with a try block that just prints out a message. After the try block, add a catch block that catches `java.io.IOException` (which obviously is not thrown by the try block). Does the code compile? If it compiles, what happens when it runs?
5. Suppose a try block throws many different subclasses of `IOException` (and no other exception types). Suppose you want to catch a few specific subclass types, such as `PrinterIOException` or `ConnectException`. All other exception types should be caught in a safety-net block. Your safety-net block can catch `IOException` or [Exception](#). The code will produce the same behavior either way, but the "Catch Blocks and instanceof" section of this chapter says that it's better to use `IOException`. Speculate on why this is true.
6. What three decisions do you have to make when creating a custom exception subclass?

Chapter 12: The Core Java Packages and Classes

Overview

At this point in your Java education, you have a solid foundation in the essential language. There are some Java features that have not been presented in this book, and won't be, but for the most part you know what you need to know to create programs consisting of classes and interfaces.

The remainder of this book will look at a large number of classes that have been written on your behalf and that you can incorporate into your code. You can use them freely. Moreover, since they are downloaded (along with the compiler and the JVM) whenever anyone downloads Java, you already have them, and you can safely assume that anyone who uses your code has the same set of classes.

We cannot possibly present them all in this book. There are well over a thousand core classes and interfaces, and many of them have specialized functionality that is of interest only to people with the same specialization. Instead of providing an exhaustive survey, we will just introduce the most important classes. Then we will show you how to learn all about the more specialized classes and interfaces. By the end of this chapter, you will have the same fundamental tools as any other Java programmer:

- An understanding of the language.
- A knowledge of certain core classes and interfaces.
- The ability to learn other core classes and interfaces as needed.
- The ability to create your own classes and interfaces, when the supplied ones don't address your needs.

The last item implies that you should use existing code wherever possible. This approach has several powerful benefits:

- The code in the core packages has been thoroughly tested.
- The code in the core packages is available immediately.
- The code in the core packages was developed at somebody else's cost (both time and money).

These benefits are offset by the principle that using an existing class to achieve an inappropriate result is generally more expensive than developing appropriate code from scratch. So a good rule of thumb is: *Use core code when you can, and develop when you must.*

The API Pages

There are more than 1,000 core Java classes, and every one of them has been described in detail by the Java creators. Unlike a lot of manufacturers' technical specifications, these descriptions are well-written, accurate, and helpful. They are provided as a set of interconnected HTML pages that you can download to your hard drive and view with the Web browser of your choice. Like Java itself, they are freely downloadable. If you have not already done so, please download them before continuing with this chapter. You can find instructions on how to do this in [Appendix A](#).

The API pages do a fine job of presenting each class in detail. In fact, they do such a good job that there is no need to duplicate their effort in this chapter. Instead, we will begin by showing you how to use the API pages. After that, we will mostly give you just an overview of the classes and methods, while encouraging you to use your new API skills to look up details when you need them.

Digression: A Personal Anecdote

Quite a while ago, before the birth of the World Wide Web, I worked for a company that made computers. These computers used a programming language that was a bit like C, but it was object-oriented. In addition to the language, there were a number of classes that supported I/O, graphical user interfaces, math, and so on. If you saw it today, you would probably be reminded of Java, but with fewer supplied classes.

There were several dozen of these classes. Their documentation consisted of two manuals that listed the classes in alphabetical order. For each class, the manuals listed the inheritance hierarchy, the data, and the methods.

Those of us who wrote programs for this system each had our own copy of the manuals. You could tell how long someone had been working there by the shape their manuals were in. Those books took a beating. We were always flipping back and forth. If I wanted to remind myself what a certain method of a certain class did, I might find that the method wasn't explained where I expected an explanation, because the class I was reading about inherited the method from its superclass. So I would look up the superclass (which might be in the other volume), and I would see that it returned an object reference, and I would have to look up that object's explanation because I hadn't seen it before.

I flipped a lot of pages because a lot of information for one class was (quite rightly) presented in the description of a different class. For example:

- The class's superclass
- The type of a non-primitive variable
- The type of a non-primitive method argument
- The type of a non-primitive method return value
- Any class, method, or variable mentioned for any reason in the description I was reading

A set of HTML documents would have eliminated all that page-turning. The only problem was, this was 1988 and there was no HTML. The Web was just a glimmer in the eyes of a few people in Switzerland, and hypertext was an idea that wasn't discussed much outside of universities. We programmers would often wonder, "Couldn't we fix it so I could read all this on my screen, and somehow click on names of classes and data and methods to read their explanations?" But we didn't invent the World Wide Web.

Fast-forward to right now. We don't have a few *dozen* classes; we have more than a *thousand*. If you printed their explanations in books, how many yards of shelf space would be required? How long would it take to look something up? Fortunately, it's all in HTML files, and fortunately, someone invented the Web. But it wasn't me.

Starting at the Top

To get the most out of this section, read it in front of your computer. You will be invited to look at various API pages. If you haven't yet downloaded and installed them, do it now!

[Appendix A](#) suggested that you download the Java documentation into `j2sdk1.4.1_02\docs` (depending on your operating system, the path separator might be a forward slash). In the `docs` directory, there is a directory called `api`. Display the contents of that directory. Double-click on the icon for `index.html`. Your browser will appear, displaying a fairly large page with 3 frames. This is the index. It is your entry point into the dozens of millions of bytes of information that are your electronic documentation.

The API pages are copyrighted, so we can't show you a picture of the index. Its structure is shown in [Figure 12.1](#).



Figure 12.1: Structure of the API index

The index has 3 frames:

- The packages frame
- The classes frame
- The details frame

Look at the packages frame in the upper-left corner of your browser. It is a list of all the core Java packages. Subpackages are also listed. Notice that the 2nd package in the list is `java.awt`, which is followed by its 10 subpackages.

The remainder of the left edge of the page is occupied by the classes frame. Initially, all the classes of all the core packages are displayed. When you click on an individual package in the packages frame, the classes frame displays the contents of the selected package. [Figure 12.2](#) shows the structure of the classes frame.

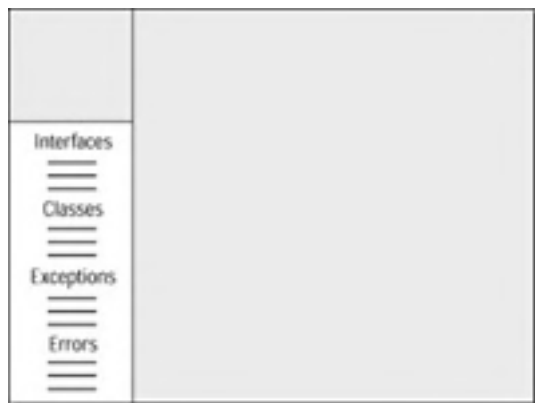


Figure 12.2: Structure of the classes frame

The classes frame shows all the interfaces, classes, exceptions, and errors of the selected package. (If you haven't yet selected a package, or if you select All Classes in the packages frame, you see everything for all packages.) Each interface, class, exception, and error is a link.

We haven't discussed errors in this book. They are like exceptions, but they indicate something deeply wrong with a program. As a programmer, you should avoid throwing or catching errors. So for the remainder of this book, we will continue to ignore their existence.

Try it. In the packages frame, click on `java.awt`. (We will spend the last three chapters of this book learning how to use this package.) The classes frame shows that `java.awt` has a large number of interfaces, a very large number of classes, a few exceptions, and one error.

Now go back to the packages frame, scroll down a bit, and click on `java.lang`. Click on the link for the `Boolean` class, which is the first link in the list of classes. The details frame displays a complete explanation of the class.

The class description is quite long, even for a simple class like `java.lang.Boolean`. It is divided into 3 sections, as shown in [Figure 12.3](#).

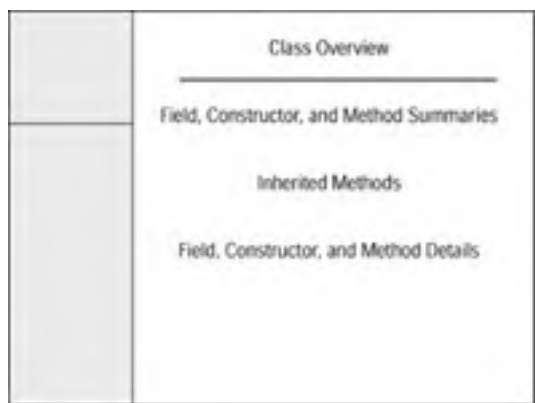


Figure 12.3: Class description

The class overview presents the class name, its inheritance hierarchy, and a text description. All elements of the inheritance hierarchy are links, so it's easy to look up a class's superclass.

The field, constructor, and method summaries are the really useful part of this frame. [Figure 12.4](#) shows their structure.

Field Summary	
data type	field name description
data type	field name description

Constructor Summary	
constructor signature	constructordescription
constructor signature	constructordescription

Method Summary	
return type	method signature description
return type	method signature description

Figure 12.4: Field/constructor/ method summaries

Each field, constructor, and method is listed alphabetically, along with a brief description. To see a more detailed description, click on the name of the field, constructor, or method. The name is a link to the position, further down on the same page, of the detailed description. Any class name in the detailed description is a link to the API page for that class. Try it. In the method summary section, click on the link for `toString()`. You see an explanation that is somewhat deeper than that brief one that appeared in the summary. Any exceptions thrown by any constructor or method are listed in the detailed description.

Any field, constructor, method, or class mentioned anywhere in the details frame is itself a link. The return type of the `toString()` method is `String`. Click on one of the occurrences of this word. The details frame displays the description of the `java.lang.String` class.

Between the summary and detail sections is a short section that lists all the methods that the class inherits from all its parent superclasses.

Typically, a session with the API pages goes like this. You want to look up a particular method of a particular class, for any of the same reasons you would look up a word in a dictionary. You want to know:

- How to spell it.
- How to use it.
- What it means.

For a method, you probably want to know one of the following:

- Its spelling.
- Its return type.
- Its argument list.
- What it does.
- What exceptions it throws.

You begin your session by scrolling through the packages frame until you find the right package. You click on it, so that the classes frame displays the contents of the package. You scroll through the classes frame until you find the class you want. You click on the class link to make the details frame display the class description.

Now you scroll through the alphabetical list of methods until you find the one you want. The summary information might be enough. If not, you click on the method name to view the detailed description.

If you're looking for the method's return type or argument list, you might find yourself looking at the name of a class that you don't recognize. No problem. Click on the class name (it's a link) and read its API page.

The API pages contain more information than is presented here, but this is enough to get you going. If you are curious about the additional API information, a good place to start is the very top of any class description page. Click on the Use, Tree, or Index link.

Deprecated Methods

Occasionally, an API page might tell you that a certain method is *deprecated*. A deprecated method is one that was introduced in an early revision of Java, and since then has been replaced with something more robust, modern, bug-free, or trustworthy. You are strongly cautioned not to use anything that is deprecated. Sun reserves the right to remove anything deprecated from future revisions.

Ordinarily, Java is *backward-compatible*. This means that if you write Java code that compiles successfully and runs correctly with the current revision of Java, your code will still compile successfully and run with the same behavior as before in any future revision of Java. But if you use deprecated methods, you no longer get backward-compatibility. If one of the methods you call has been removed, your code will no longer work.

Team LiB

← PREVIOUS NEXT →

The *java.lang* Package

The core Java classes fall into three broad categories. There are classes that support specific tasks, such as database access or graphical user interface (*GUI*) creation. These classes are organized into packages. For example, database support is in the `java.sql` package, while GUI infrastructure is in the `java.awt` package and its many subpackages. Another category is classes that are generally useful, no matter what kind of program you are writing. Most of these appear in the `java.util` package. The third category is classes that are essential to the operation of any program. These are to be found in the `java.lang` package. Let's begin with a look at a few of them.

The classes and interfaces in `java.lang` are so important that they are imported in all source code automatically. It is as if the compiler inserted the following line into any source:

```
import java.lang.*;
```

We will not be looking at all the classes in `java.lang`. Again, the purpose of this book is not to tell you everything there is to know about every class in the package. Instead, we will just look at a few of the most important classes. Since you know how to read API pages, you know how to find out about the others.

The *java.lang.String* Class

We will start with `String`. You have been aware of this class ever since [Chapter 2](#), when you first looked at applications and saw the following:

```
public static void main(String[] args)
{
    . . .
}
```

Now you know that this is a method declaration, and no doubt you've guessed that the method takes a single argument whose type is an array of something called `String`.

The `String` class contains an ordered sequence of characters, representing a piece of text. The text that an instance contains is specified as a constructor argument. The class is *immutable*. This means that after an instance is constructed, its contents cannot be changed. So if an instance of `String` initially contains the text "Click here to select a color", it will contain that text throughout its lifetime.

This class is unique, in that there are two ways to create an instance. One way, of course, is to call a constructor. The second way, which is unique to the `String` class, is to use a *literal string*. A literal string is text enclosed in double-quotes, like this:

```
"I am a literal string."
```

When the compiler encounters a literal string, it generates code that creates an instance of `String` to represent the text in quotes. (Actually, the situation is a bit more complicated than that, but we don't need to go into detail here.) We have often used code with the following format:

```
System.out.println("value is " + x);
```

Now you know that the text between the quotes is a literal string. Later in this chapter, you will see what is really going on when the literal string is added to `x`. For now, you know that whatever else might be happening in the line of code, execution involves the creation of an instance of `String` that represents the text in quotes.

The shortest literal string is

```
""
```

This string has zero characters, but it is still an object that exists, and you can call any methods of the `String` class on it. It is called the *empty string*.

The easiest way to create a `String` instance that contains a particular run of text is like this:

```
String s;
s = "To be, or not to be";
```

Or simply:

```
String s = "To be, or not to be";
```

There are 11 different versions of the `String` constructor. Here we will only look at one of them. (Later in this chapter you will learn how to look for information about the rest, so you will be able to choose the best one for any situation.) The simplest constructor is

```
public String(String s)
```

This constructor takes a single argument, which is a reference to another string. The new object is an exact replica of the argument. Even though this is the simplest form of the `String` constructor, it isn't often used because you have the option of using a literal string instead. For example, the following two lines are (almost) equivalent:

```
String s = new String("abcde");
String s = "abcde";
```

The second version is obviously easier to type.

The `String` class has a large number of methods. Here we will present a few of the more interesting ones. We start with `toUpperCase()` and `toLowerCase()`. The `toUpperCase()` method converts all lowercase characters in a string to uppercase. The `toLowerCase()` method converts all uppercase characters in a string to lowercase. Here is an example:

```
String s = "Mm";  
String upper = s.toUpperCase();  
String lower = s.toLowerCase();  
System.out.println(upper);  
System.out.println(lower);
```

The output of this code is

```
MM  
Mm
```

These methods, and indeed all `String` methods that seem to modify a string, do not change the original string. That would be impossible, because strings are immutable. After they are constructed, they cannot be changed. Instead, `toUpperCase()` and `toLowerCase()` create and return a new string object. To prove this, you can modify the code example:

```
String s = "Mm";  
String upper = s.toUpperCase();  
String lower = s.toLowerCase();  
System.out.println(upper);  
System.out.println(lower);  
System.out.println(s);
```

The output of this code is

```
MM  
mm  
Mm
```

Notice the last line, which prints out the unscathed original string.

The situation can get confusing when a single reference is reassigned, as in the following example:

```
String s = new String("UPPER : lower");  
System.out.println("BEFORE: " + s);  
s = s.toUpperCase();  
System.out.println("AFTER: " + s);
```

Here the single reference `s` refers first to the original string, and a little later to the new uppercase string. It *looks like* there is only a single string object involved, especially when you look at the output:

```
BEFORE: UPPER : lower  
AFTER: UPPER : LOWER
```

But there are actually two objects, although there is only one reference. When the reference is reassigned (`s = s.toUpperCase()`), the original string object might get garbage-collected. This would happen if there were no other references to the original object.

This might seem overly complicated when all you wanted to do was convert a string to upper- or lowercase, but it is always important to bear in mind the difference between references and objects, and to know exactly what references are pointing to what objects at every point in your code.

The `StringLab` animated illustration demonstrates strings in moving pictures. To run the animation, type `java strings.StringLab`. You see a window with two code statements, as shown in [Figure 12.5](#).

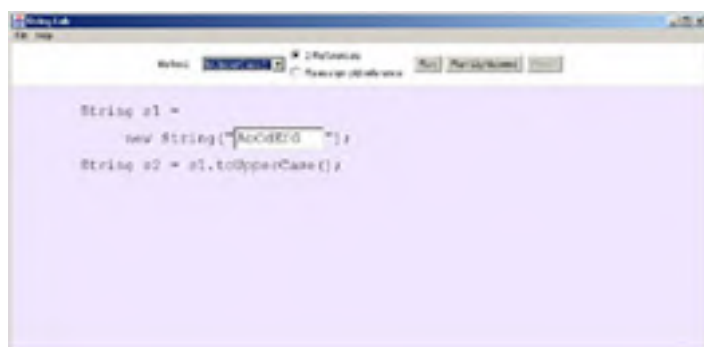


Figure 12.5: StringLab

The first statement creates a new instance of `String`. You can type anything you like to provide the text. The second statement calls a method, `toUpperCase()` or `toLowerCase()`, on the string. Use the controls at the top of the window to choose the method that will be called. You can also use the controls to choose between using two references or reassigning a single reference. (Notice how the code in the main window changes when you change this option.) As usual, click `Run` to see the animation, or click `Run Lightspeed` to skip the animation and see the final result. Be sure to watch the full animation with `Reassign Old Reference` selected so you can see the original string being garbage-collected.

[Figure 12.6](#) shows the result of converting to uppercase and using two references. [Figure 12.7](#) shows the result of converting to lowercase and reassigning the reference.



Figure 12.6: StringLab: uppercase, 2 references



Figure 12.7: StringLab: lowercase, 1 reference

Now that you understand strings and methods, we can move quickly through a list of other *String* methods:

- `trim()` Removes blank spaces from the start and end (but not the middle) of the executing string object.
- `substring(int n)` Returns a portion of the executing string object. The substring consists of the run of characters beginning at position *n* (where 0 is the first character) and ending at the end of the executing string object.
- `concat(String s)` Appends *s* to the executing string object.

The return types are all *String*. These descriptions are deliberately brief. Detailed explanations are available to you in the API pages.

Here are some *String* methods that return information about the executing string object. The return types vary, so they are included in the list:

- `boolean equals(String s)` Returns `true` if *s* and the executing string object contain identical text.
- `boolean equalsIgnoreCase(String s)` Returns `true` if *s* and the executing string object contain identical text, ignoring uppercase and lowercase distinctions.
- `char charAt(int n)` Returns the *n*th character in the executing string object.
- `int length()` Returns the length of the executing string object.
- `boolean startsWith(String s)` Returns `true` if the executing string object begins with string *s*.

Here is a method whose argument is a string. The method prints out every character of the argument on its own line:

```
void printChars(String s)
{
    int length = s.length();
    for (int i=0; i<length; i++)
    {
        char c = s.charAt(i);
        System.out.println("Char #" + i + " is " + c);
    }
}
```

Note the use of the `length()` and `charAt()` methods. Here is the output when the method is called with argument *Alligator*:

```
Char #0 is A
Char #1 is l
Char #2 is l
Char #3 is i
Char #4 is g
Char #5 is a
Char #6 is t
Char #7 is o
Char #8 is r
```

The only tricky method presented here is `equals()`. It is easy to understand what it does, provided you don't get misled by the name. The `equals()` method does not check if the argument and the executing string object are the same object.

Instead, it checks whether the two objects both represent identical sequences of characters. To check if the argument and the executing string are the same object, use the `==` operator, as demonstrated in the following example:

```
String s1 = new String("aa");  
String s2 = new String("aa");  
String s3 = s2;  
if (s1.equals(s2))  
    System.out.println("s1.equals(s2): YES");  
if (s1 == s2)  
    System.out.println("s1 == s2: YES");  
if (s2 == s3)  
    System.out.println("s2 == s3: YES");
```

The output is

```
s1.equals(s2): YES  
s2 == s3: YES
```

Figure 12.8 shows the references and objects of this example.

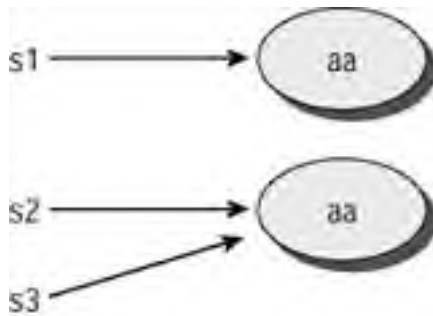


Figure 12.8: String references and objects

We say that the `==` operator checks for *reference equality*, which means it checks if two references point to the same object. The `equals()` method checks for *object equality*, which means it checks if two objects contain equal data. This distinction is very important in object-oriented programming.

Command-Line Arguments

Every Java application has a main method that begins like this:

```
public static void main(String[] args) . . .
```

Of course, you can call the method argument anything you like, but `args` is the conventional name. The array contains the application's *command-line arguments*. These are everything the user has typed into the command line that invoked the application, except for the following:

- `java`
- The application class name
- Any arguments for the JVM

So if you have an application class called `database.Backup`, and you run it by typing `java database.Backup network local greebo 1234`, the `args` array will look like Figure 12.9.

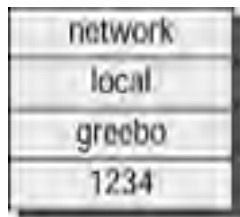


Figure 12.9: Command-line arguments

Note that the last component in the array is a string. It is not a number, even though it looks like one because it consists entirely of digit characters. Later you will learn how to convert an all-digit string into an int.

One job of the `main()` method is to check the command-line arguments and take appropriate action. For example, a `main()` method might have the following code:

```
public static void main(String[] args)
{
    for (int i=0; i< args.length; i++)
    {
        if (args [i].equals("help"))
        {
            printHelpMessage();
            break;
        }
    }
}
```

We assume the existence of a `printHelpMessage()` method that prints out an explanatory message on how to use the program. The code scans the `args` array. If the array contains any occurrence of "help" (that is, if someone typed "help" anywhere on the command line), the explanatory message is printed.

Some command-line arguments are intended for the JVM and do not get passed into the application. To see a list of JVM arguments, type `java -help`. One such argument is `-verbose`. When you run a program with this argument, you get output from the JVM about such activities as class loading. If you invoke an application by typing `java -verbose database .Backup network local greebo 1234`, the `-verbose` argument will be consumed by the JVM and won't be passed on to the application. So the application's `args` array will still be as shown in [Figure 12.9](#), but the output will include the verbose messages from the JVM.

The `java.lang.Object` Class

The `Object` class is the ultimate superclass of all other Java classes. It has about a dozen methods, many of which support advanced functionality that is beyond the scope of this book. But it does have two methods that everyone should know about: `equals()` and `toString()`. Since every other class extends `Object` (directly or indirectly), every other class inherits these methods. You are not likely to create instances of `Object`, but you are very likely to use the `equals()` and `toString()` methods that other classes inherit or override.

The first method we will look at is `equals()`. You already know about the implementation provided by the `String` class, and that it checks for object equality and not reference equality. This distinction is maintained throughout the core Java classes: Any implementation of `equals()` in any core class checks for object equality rather than reference equality. So `thisRef.equals(thatRef)` will return `true` if the two references point to objects that contain equal variables. (Of course, if the two references point to the same object, the call will also return `true`.)

For example, the `java.awt.Point` class represents a point in two-dimensional space. It has variables called `x` and `y`, which hold the horizontal and vertical location of the point. If `p1` and `p2` are both references to instances of this class, you can check for object equality by calling either

```
if (p1.equals(p2)) ...
```

or

```
if (p2.equals(p1)) ...
```

The `equals()` method of class `Point` returns `true` if `p1.x` equals `p2.x` and `p1.y` equals `p2.y`.

Not all core classes provide their own implementations of `equals()`. If you want to know if a class you are interested in provides an implementation, you should look at the class's API page. Beware of classes whose API pages do not document an `equals()` method. The class might inherit the method from `Object`, and that version is not very useful.

Now we will turn to the extremely useful `toString()` method. It is public, it returns a `String`, and it has no arguments, so its declaration looks like this:

```
public String toString()
```

This method is intended to print out a useful message that includes information about the values of the executing object's variables. The version provided by the `Object` class isn't very informative. In fact, it's downright cryptic. But every one of the core Java classes overrides `toString()` with a version that provides useful information.

For example, there is a class called `java.awt.Color` that represents a color. The class has `int` variables called `red`, `green`, and `blue`, which contain the amounts of red, green, and blue light that make up the represented color. Their values can range from 0 through 255, and they are specified in the class's constructor.

Suppose you have a long intricate program with an instance of `Color`, referenced by variable `foreground`, that doesn't look right. (Colors are used extensively in visual programming, which we will look at in the last 3 chapters of this book.) It would be helpful if you knew exactly what the red, green, and blue components of the problematic color are. That information might lead you to the source of the trouble. Thanks to `toString()`, you can easily create a line of [debug code](#) that tells you what is going on inside your program. Always delete debug code after it has served your purpose. Otherwise it will accumulate, and your program will emit lots of information that is no longer helpful.

```
Here is a debug line that prints out the puzzling color:
System.out.println("Weird color is " +
    foreground.toString());
```

The output looks something like this:

```
Weird color is java.awt.Color[r=100,g=255,b=0]
```

The output uses abbreviations instead of `red`, `green`, and `blue`, but it's clear what their values are. Now you can determine which of them are wrong and look at the code that calculates the corresponding value that is passed into the `Color` constructor.

There is an easier way to print the `toString` value of the color, or of any other object. This brings us to the topic of [string concatenation](#). "Concatenation" is another of those five-syllable words. It just means joining strings consecutively, one after another (after another, after another...). You have already used concatenation extensively, whenever you did something like

```
System.out.println("size is " + size +  
                  "and weight is " + weight);
```

Now it's time to see what's really going on between those parentheses. Look at those plus signs. Obviously they mean something other than addition. In Java, plus signs have a double meaning:

- When both items next to a plus sign are numeric, the plus sign means addition.
- When one or both items next to a plus sign are references to `strings`, the plus sign means string concatenation.

In the second context, if one of the items next to the plus sign is a string, the other item can be *anything!* It can be a primitive, another string reference, or a reference to an object of any other class. The other item is converted to a string according to the rules shown in [Table 12.1](#).

Table 12.1: String Concatenation Conversion Rules

Type	Conversion Rule
<code>boolean</code>	"true" or "false"
Primitive other than <code>boolean</code>	A reasonable string representation
<code>String</code>	The string
Object reference other than <code>String</code>	Call <code>toString()</code> on the reference

The last entry in the table means that the line

```
System.out.println("Weird color is " +  
                  foreground.toString());
```

is equivalent to

```
System.out.println("Weird color is " +  
                  foreground);
```

In other words, when you're doing concatenation, you never need to type `.toString()`.

The ConcatLab animated illustration shows concatenation in action. Run the program by typing `java concat.ConcatLab`. You will see a window with three lines of code, as shown in [Figure 12.10](#).



Figure 12.10: ConcatLab

The first statement creates an instance of `java.awt.Color`. The constructor's arguments are the three primary color values (red/green/blue) that constitute the color. They must be in the range 0-255. You can type in any valid values. Later on, you will see the color they represent.

The second statement creates an instance of a class called `Point3D`, which is not part of the core Java classes. To see the (very simple) source for `Point3D`, click on the Edit `Point3D` button. You will see the display shown in [Figure 12.11](#).



Figure 12.11: ConcatLab's `Point3D` class

The class represents a point in 3-D space, with x, y, and z coordinates. You will see a version of toString() that returns a reasonable string representation. You can edit this code. Type any text you want into the text fields. When you're finished, click on OK.

The third statement on the main window says

```
String s = "c is " + c + " and p is " + p;
```

Click on the Run button to run the animation, which shows how the four parts of the concatenated string are made. [Figure 12.12](#) shows the result of running the animation, after some custom configuration.

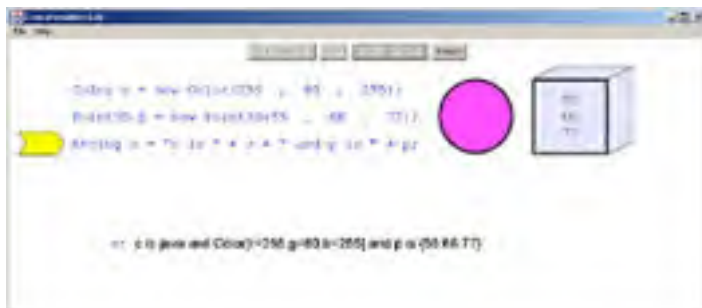


Figure 12.12: ConcatLab's Point3D class

The *java.lang.Integer* Class, and other Wrappers

The `java.lang` package has eight very simple classes called *wrappers*. A wrapper is a class whose data is a single primitive value. In other words, the primitive is "wrapped up" inside an object. The wrapper classes have names that are very similar to the corresponding primitive names. In some cases, the names are identical except for the first letter, which is always uppercase for class names and lowercase for primitive names. [Table 12.2](#) shows the wrapper class names.

Table 12.2: Wrapper Class Names

Primitive	Wrapper
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

The wrapper classes are immutable. This means that, just as with strings, the data contained in an instance doesn't change after the instance is created. The contained data is passed into the constructor, as shown below:

```
Boolean boo = new Boolean(true);  
Character cha = new Character('L');
```

```
byte b = 5;  
Byte bye = new Byte(b);  
short s = 10;  
Short sho = new Short(s);  
int i = 9999;  
Integer inty = new Integer(i);  
long n = 2222222;  
Long lonny = new Long(n);  
float f = 3.14159f;  
Float flo = new Float(f);  
double d = 1.2e200;  
Double dubby = new Double(d);
```

It might not be clear why these classes would ever be useful. You'll find out why when you learn about the `java.util` class `a` little later on. But for now, be aware that the wrapper classes all have static methods that are useful for translating strings into primitives. For example, class `Integer` has a `parseInt(String s)` method that translates a string to an int. If the string does not represent a number, the method throws `NumberFormatException`.

The following code is an application that translates its first classes to an int, multiplies that value by 39, and then prints out the result:

```
public class X39  
{  
    public static void main(String[] args)  
    {
```



```
if (args.length == 0)
{
    System.out.println("Please supply a number.");
}
else
{
    try
    {
        int n = Integer.parseInt(args[0]);
        int times39 = n * 39;
        System.out.println(args[0] + " * 39 = " +
            times39);
    }
    catch (NumberFormatException x)
    {
        System.out.println("That's not a number!");
    }
}
}
```

Note that this code doesn't create an instance of `Integer`. Instead, it calls a static method of `Integer` (in the first line of the try block) to convert the string in `args[0]` to an int. Code like this is frequently seen near the beginning of `main()` methods. Generally, an application that has numeric command-line arguments can't get very far until it converts the argument strings to numeric primitives.

The `java.lang.System` Class

The `java.lang.System` class contains a hodgepodge of methods, most of which involve advanced functionality related to the JVM. This brief section will only cover one of the class's methods. First, let's take a moment to look at two of its static variables: `out` and `in`.

You have already used `System.out` extensively. Whenever you used

```
System.out.println(...);
```

You were making a call to the `println()` method of the `System.out` object.

The `println()` method is heavily overloaded. All of the versions of the method take a single argument. One version, which you have been using throughout this book, takes a string argument. The string is printed out, followed by a *newline* character. The newline character is not displayed. Instead, it moves the cursor position to the beginning of the next line.

Other versions of `println()` take args that are bytes, shorts, booleans, and so on. These versions convert their arguments to strings and then print them out, followed by a newline character.

Perhaps the most commonly used method of `java.lang.System` is `exit()`. This method causes the JVM to terminate, thus ending the current application immediately. The method takes an int argument called the *exit code*. Typically, 0 is used to indicate a normal termination, while a non-zero value indicates that termination was caused by an error condition.

Some operating systems are able to run sequences of programs, where the exit code of one program is used to control the operation of the next program. This is highly system-dependent and not relevant to an introductory Java book, but you need to know what exit status codes are because you need to pass an argument into every `System.exit()` call. You won't go wrong if you use 0 to mean normal termination and a small non-zero value to mean abnormal termination.

The following code is a rewrite of the previous example, using `System.exit()` to terminate execution.

```
public class X39RevB
{
    public static void main(String[] args)
    {
        if (args.length == 0)
        {
            System.out.println("Please supply a number.");
            System.exit(1); // Non-zero exit code
        }
        else
        {
            try
            {
                int n = Integer.parseInt(args[0]);
                int times39 = n * 39;
                System.out.println(args[0] + " * 39 = " +
                    times39);

                System.exit(0);
            }
            catch (NumberFormatException x)
            {
                System.out.println("That's not a number!");
                System.exit(2);
            }
        }
    }
}
```

The `System.exit(0)` call at the end of the try block isn't actually necessary. When `main()` finishes, the JVM shuts down anyway, with an exit code of 0.

The `java.lang.Math` Class

The `java.lang.Math` class is the least object-oriented of all the core Java classes. All of its methods are static, so you never need to create an instance. In fact, you aren't *allowed* to create an instance (see Exercise 3 for details).

The class has dozens of methods. Instead of listing them all, here's a short sampler. Consult the API page for the complete list.

```
int min(int a, int b) Returns the lesser of a and b
int max(int a, int b) Returns the greater of a and b
double sin(double angle) Computes the sine of an angle
double cos(double angle) Computes the cosine of an angle
double tan(double angle) Computes the tangent of an angle
double pow(double a, double b) Returns a raised to the power of b
double random() Returns a random number that is >=0 and <1
```

And so on. All trigonometry methods use radians, not degrees, for expressing angles. All methods that do intense calculation have return types of `double`.

The `random()` method can be used to generate a random double in any range. For example, to generate a random double that is ≥ 0 and < 30 , just use `30 * Math.random()`. To generate a random double that is ≥ 10 and < 40 , just use `10 + (30*Math.random())`.

The following code generates 100 random numbers that are ≥ 0 and < 50 . Then it computes the area of a circle whose radius is the random number. The code keeps track of, and prints out, the largest area:

```
public class RandomAreas
{
    public static void main(String[] args)
    {
        double maxArea = 0;
        for (int i=0; i<100; i++)
        {
            double radius = 50 * Math.random();
            double area = 3.24159 * radius * radius;
            if (area > maxArea)
                maxArea = area;
        }
        System.out.println("Biggest area = " + maxArea);
    }
}
```

The `lang` class defines two public final static double variables, `PI` and `E`. They contain very precise values for these mathematical constants, accurate to 20 digits to the right of the decimal point. So you never need to memorize, look up, or type in either of these values. That's a good thing, because you might accidentally type in a wrong value that could throw off all your subsequent calculations. In fact, that's what happened in this example. In the line that begins `double area =`, the `2` should be a `1`. A better line would be

```
double area = Math.PI * radius * radius;
```

You can make the code a little shorter by replacing these lines:

```
if (area > maxArea)
    maxArea = area;
```

With this single line:

```
maxArea = Math.max(area, maxArea);
```

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. In the beginning of this chapter you learned that a good rule of thumb is to use core code when you can and develop original code when you must. Because Java is an object-oriented language, you have a third option, which combines reusing existing code with creating your own. You learned about this option in an earlier chapter. What is it?
2. If you write code that calls a deprecated method of one of the core Java classes, what valuable feature of Java can you no longer rely on?
3. Suppose you are reading someone else's code and you come across the following lines:

```
Stack myStack = new Stack(); // java.util package
myStack.setSize(100);
```

You decide to look up `setSize()` in the APIs. The comment kindly tells you that class `Stack` is in package `java.util`, so you click on `java.util` in the packages frame, and then you click on `Stack` in the classes frame. You find yourself looking at the class description. You scroll down to the method summaries, and you don't see `setSize` anywhere.

How should you proceed?

4. In the section on the `String` class, you learned about the `startsWith(String s)` method, which returns `true` if the executing string object begins with the argument string `s`. It stands to reason that there should be a similar method that tells you whether the executing string object ends with a specified string. Look at the API page for `java.lang.String` and see if such a method exists.
5. What happens when you try to compile and execute the following application?

```
public class Ch12Q5
{
    public String toString()
    {
        return "I am an instance of Ch12Q5.";
    }
    public static void main(String[] args)
    {
        Ch12Q5 thing = new Ch12Q5();
        System.out.println(thing);
    }
}
```

6. What happens when you try to compile and execute the following application?

```
class Ch12Q6
{
    String toString()
    {
        return "I am an instance of Ch12Q6.";
    }
    public static void main(String[] args)
    {
        Ch12Q6 thing = new Ch12Q6();
        System.out.println(thing);
    }
}
```

7. Look up the explanation of the `equals()` method on the API page of class `java.lang.Object`. The explanation is a bit wordy, but see if you can figure out what it does. (Focus on the last sentence, just before the "Parameters" section.) What is the technical term for what the method does? (Hint: It was introduced in this chapter.)
8. You're not allowed to construct an instance of the `java.lang.Math` class. What happens if you try?
9. The following code models the behavior of a familiar piece of equipment that is used in many games throughout the world. What is the piece of equipment?

```
long rand = 1 + Math.round(Math.random() * 5);
```

Chapter 13: File Input and Output

In [Chapter 12](#), you saw a few of the core Java packages and classes. You also learned that creating successful Java programs involves both writing your own code and using preexisting classes. This chapter will cover the fundamentals of reading and writing disk files. It will take advantage of several core classes in the `java.io` package. This will be your first look at making extensive use of core classes.

Files As Sequences of Bytes

In [Chapter 1](#), you saw that a computer's memory is a clump of tiny circuits in which voltages represent 0s and 1s. It doesn't take a degree in electrical engineering to know that when you turn off a circuit's power, the voltages go away. No more 0s, no more 1s.

Disks are like computer memory in the following sense: A disk is a collection of tiny "somethings" that can be in one of two possible states. The surrounding electronics, and the software that controls the surrounding electronics, interpret the two states as representing 0 or 1. With a hard disk, the states are microscopic magnetic fields that can point in either of two directions. With a CD-ROM or DVD, the medium is filled with microscopic regions that either do or do not block light. Aside from the underlying physics, the main difference between disks and memory is that disks remember what is stored in them, even after the power goes off.

To make the rest of this discussion more clear, let's use the term *RAM* to mean ordinary computer memory, as distinct from disks, which are also a kind of memory. RAM is an acronym for Random Access Memory. It's a cool-sounding acronym, but you may be wondering what's so random about RAM. "Random" relates (distantly) to the amount of time it takes to read data out of memory or to write data into memory. It takes exactly the same amount of time (less than one millionth of a second) to read any byte in the circuit. Writing might take slightly longer than reading, but writing any byte takes exactly the same amount of time as writing any other byte. So you can pick any two bytes at random, and they can be read in the same amount of time, or written in the same amount of time, as each other.

Disks are not random access devices. At any moment, some parts of the disk data can be read more quickly than others. This is because the disk is rotating. If you want to read some data, you have to wait until it has rotated into position next to the disk's reading or writing hardware, which does not rotate. If you're lucky, the data will be just about rotated into position. If you're out of luck, the data will have just rotated out of position, and you will have to wait until the disk makes another revolution.

So you see that RAM and disks have very different mechanical and physical properties, but they both can be treated as storing ordered sequences of 0s and 1s.

As with RAM, you think of disks as being organized into bytes, each byte having a unique position. As with RAM, you would find it impossibly limiting if you had to think exclusively in terms of bytes. As with RAM, you use groups of disk bytes to encode higher-level multi-byte information. But unlike RAM, the first step in learning how to do disk input and output is to learn how to read and write pure bytes. That is where we will begin.

Writing and Reading Bytes

Document files don't really contain text. Image files don't really contain pictures. MP3 files don't really contain music, and MPEG files don't really contain movies. They all contain bytes, because all files just contain bytes. The bytes encode information; they are decoded by software appropriate to the encoded content. This is why filename extensions are so important. They tell the computer what decoding software to use. If you take an image file that encodes a really beautiful picture and you change filename extension to .mp3, it's probably going to sound terrible.

All files are sequences of bytes. Before we look at decoding and encoding the information represented by the files, you need to learn how to write and read plain ordinary bytes. You will make extensive use of two of the classes in the `java.io` package:

- `FileOutputStream`
- `FileInputStream`

A file output stream writes bytes to a file; a file input stream reads bytes from a file. Our purpose here is not to present both classes in their entirety. Here you will learn more than enough to be able to use them well. Whenever you want to complete your understanding, you can refer to the API documentation.

Both classes have constructors with `String` arguments, where the string specifies the name of the file. On Windows machines, the *file separator* (the character that goes between elements in a full pathname) is a backslash, and that can lead to problems. So let's begin with a digression on dealing with backslashes.

Backslashes in Filenames

If you want to write bytes to a file in the current working directory called `xyz`, you can construct a file output stream like this:

```
FileOutputStream fos;  
fos = new FileOutputStream("xyz");
```

Of course, you can create a file output stream in a similar way. Ignoring for the moment the issue of what you can actually do with those streams, you have to deal with the question of what happens when you want to specify a full pathname on a Windows system. For example, what if you want to write to a file whose full pathname is `C:my_files\photos\abc`? The following code will not do what you want:

```
FileOutputStream fos;  
fos = new FileOutputStream("C:my_files\photos\abc");
```

Surprisingly, this code will not compile! The compiler error says that there is an invalid escape character, whatever that means.

Actually, the problem has nothing to do with file output streams. It has to do with backslashes in literal strings. You would get the same compilation error if you tried the following:

```
String s = "C:my_files\photos\abc";
```

In [Chapter 2, "Data,"](#) you saw that certain characters (most notably the newline and tab characters) are represented by escape codes, `\n` for newline and `\t` for tab. Those codes can also be embedded in literal strings. For example, the following code prints some numbers, separated by tabs, on two lines:

```
String s = "123\t456\t789\n987\t654\t432";  
System.out.println(s);
```

You can see that the backslash character has special meaning to the Java compiler. In literal strings and chars, backslash means, "Ignore me and treat the next character as a special code." If you just want a simple ordinary backslash in a literal string or char, you have to use a double backslash. For example, to print out the word "hello" followed by a backslash, you have to do the following:

```
System.out.println("Hello\\");
```

Note the second backslash. Only one backslash is printed.

So you can see that

```
String s = "C:my_files\photos\abc";
```

won't compile, because `\p` and `a` are not valid escape codes. It's a good thing they aren't. The following code compiles, but with an unexpected result:

```
String s = "C:my_backup\temporary\news";
```

So if you are writing file access code for a Windows machine, you always have to remember to use double backslashes for file separators, like this:

```
FileOutputStream fos;  
fos = new FileOutputStream("C:my_files\\photos\\abc");
```

Now that you know how to specify filenames, we can move on to writing to files.

Writing Bytes

To create a file full of bytes, you have to do three things:

1. Construct an instance of `FileOutputStream`.
2. Write the bytes.
3. Close the stream.

The following application creates a file called "xyz" in the current directory, writes 10 bytes, and then closes the file:

```
1. import java.io.*;
2.
3. public class Write10Bytes
4. {
5.     public static void main(String[] args)
6.     {
7.         FileOutputStream fos;
8.         fos = new FileOutputStream("xyz");
9.         for (int i=0; i<10; i++)
10.            fos.write(i);
11.        fos.close();
12.    }
13. }
```

Line 8 constructs the output stream. Line 10, which executes 10 times in the `for` loop, writes the bytes. It looks like line 10 actually writes ints, because `i` is an int, but the `write()` method actually only writes the low-order 8 bits of its argument. Line 11 "closes" the stream. *Closing* releases certain hidden operating system resources that the stream needs in order to access the disk. After a stream is closed, it can't be written to.

Our code example will not compile, because lines 8, 10, and 11 throw exceptions. The constructor on line 8 throws `FileNotFoundException`. The `write()` call on line 10 and the `close()` call on line 11 throw `IOException`. So the code can be improved as follows:

```
1. import java.io.*;
2.
3. public class Write10Bytes
4. {
5.     public static void main(String[] args)
6.     {
7.         try
8.         {
9.             FileOutputStream fos;
10.            fos = new FileOutputStream("xyz");
11.            for (int i=0; i<10; i++)
12.                fos.write(i);
13.            fos.close();
14.        }
15.        catch (FileNotFoundException x)
16.        {
17.            System.out.println("Caught FileNotFoundException");
18.        }
19.        catch (IOException x)
20.        {
21.            System.out.println("Caught IOExn");
22.        }
23.    }
24. }
```

This code compiles, and it executes correctly. But it can be simplified a bit. `FileNotFoundException` is a subclass of `IOException`. So we can eliminate lines 13-16:

```
1. import java.io.*;
2.
3. public class Write10Bytes
4. {
5.     public static void main(String[] args_
6.     {
7.         try
8.         {
9.             FileOutputStream fos;
10.            fos = new FileOutputStream("xyz");
11.            for (int i=0; i<10; i++)
12.                fos.write(i);
13.            fos.close();
14.        }
15.        catch (IOException x)
16.        {
17.            System.out.println("Caught IOExn");
18.        }
19.    }
20. }
```

After this application runs, the current directory contains a 10-byte file named "xyz".

The Simple Output Lab animated illustration demonstrates an application that writes several bytes to a file. To run the program, type "`java io.SimpleOutputLab`". The initial display is shown in [Figure 13.1](#).



Figure 13.1: Simple Output Lab

The animation is very simple, but it will give you a good graphical image of the relationships between the data, the output stream, and the file. [Figure 13.2](#) shows the animation in progress.

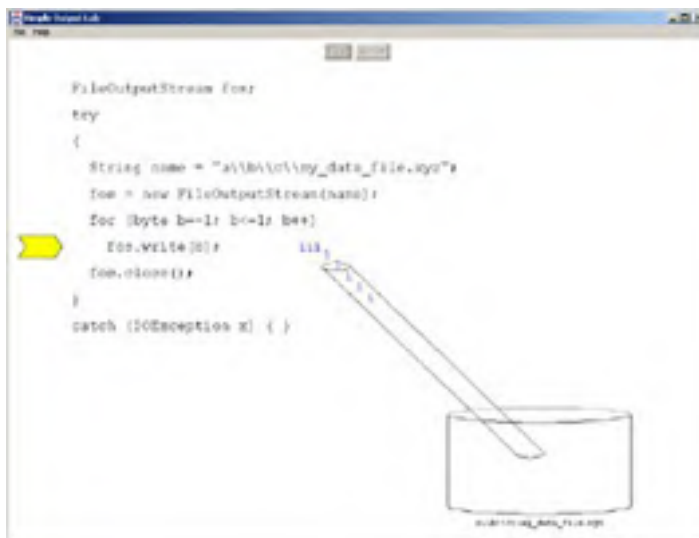


Figure 13.2: Simple Output Lab in progress

Reading Bytes

Reading bytes is almost exactly like writing bytes. You still have to do three things:

1. Construct an instance of `FileInputStream`.
2. Read the bytes.
3. Close the stream.

The following application reads back the file that was created in the [previous section](#):

```
1. import java.io.*;
2.
3. public class Read10Bytes
4. {
5.     public static void main(String[] args)
6.     {
7.         try
8.         {
9.             FileInputStream fis;
10.            fis = new FileInputStream ("xyz");
11.            for (int i=0; i<10; i++)
12.            {
13.                int theByte = fis.read();
14.                System.out.println(theByte);
15.            }
16.            fis.close();
17.        }
18.        catch (IOException x)
19.        {
```

```
20.         System.out.println("Caught IOExn");
21.     }
22. }
23. }
```

Line 8 creates the input stream, line 12 reads the bytes, and line 14 closes the stream. Line 12 prints out the bytes that were read, each one on its own line.

Note the strange variable on line 11. It is called `theByte`, but it is an `int`. The `read()` method of class `FileInputStream` reads a byte from the disk, but returns an `int`. Usually, the high-order 24 bits of the returned `int` are all 0s; the low-order 8 bits are the byte that was read from the disk. However, if the input stream has already read all the bytes in its file, the next `read()` call will return the `int` value -1. Recall that this value consists of 32 1's. This is distinct from the byte value of -1, which consists of eight 1's. If a file input stream reads such a byte from its file, the return value will have 1s in its low-order eight bits, and 0s in its high-order 24 bits. So there is no danger of confusing a byte read from the file whose value happens to be -1 with the `int` that signals that there is no more data in the file. [Table 13.1](#) makes this clear.

Table 13.1: Byte -1 vs. Int -1

byte -1, returned as an int	int -1, signaling end of file
00000000 00000000 00000000 11111111	11111111 11111111 11111111 11111111

You can use the special return value when you don't know the length of the file you are reading. In this example, suppose you don't know that the file contains 10 bytes. As you learned in [Chapter 5](#), when you don't know how many times the loop will execute, it's time to use a while loop:

```
import java.io.*;

public class Read10Bytes
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fis;
            fis = new FileInputStream ("xyz");
            while (true)
            {
                int theByte = fis.read();
                if (theByte == -1)
                    break;
                System.out.println(theByte);
            }
            fis.close();
        }
        catch (IOException x)
        {
            System.out.println("Caught IOException");
        }
    }
}
```

This application generates exactly the same output as the previous version, but this time there is no need to know the size of the file. This version can handle a file of any size.

The Simple Input Lab animated illustration demonstrates an application that reads several bytes from a file. To run the program, type "`java io.SimpleInputLab`". The animation is very simple, but like `SimpleOutputLab`, it will give you a good graphical image of the relationships between the data, the input stream, and the file. [Figure 13.3](#) shows the animation in progress.

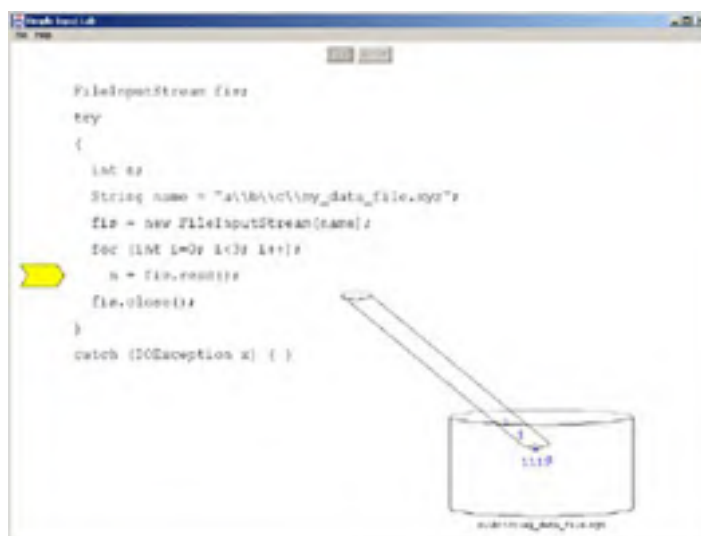


Figure 13.3: Simple Input Lab in progress

Team LIB

◀ PREVIOUS

NEXT ▶

Writing and Reading Data

At this point, you might be asking yourself, "When would I ever want to write or read bytes?" After all, one of the huge disadvantages of SimCom, as compared to the JVM, is that it deals only in bytes, while Java supports eight primitive data types and limitless class types.

The answer, fortunately, is that you never have to write or read bytes if you don't want to. You still have to create file input and output streams, and you still have to close them when you're done using them, but you don't have to write to them or read from them. Not directly, anyway. The writing and reading can be done by two very useful classes in the java.io package:

- `DataOutputStream`
- `DataInputStream`

The constructor for `DataOutputStream` takes a single argument. This argument is not the name of a file. Instead, it is a reference to a file output stream. Data written to a data output stream gets chopped up into bytes, which the data output stream passes to its file output stream. The technique of connecting streams together is called *chaining*. [Figure 13.4](#) shows a data output stream chained onto a file output stream.

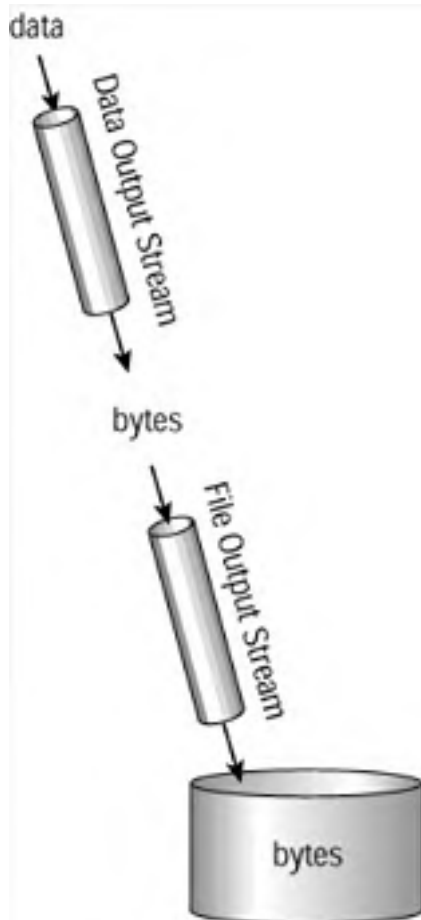


Figure 13.4: Output chaining

`DataOutputStream` has a large number of methods that chop up data and deliver bytes to the next stream in the chain. Here we will discuss nine of these methods:

- `writeBoolean(boolean boo)`
- `writeByte(int b)`
- `writeShort(int s)`
- `writeChar(int c)`
- `writeInt(int i)`
- `writeLong(long n)`
- `writeFloat(float f)`
- `writeDouble(double d)`

■ writeUTF(String s)

It is obvious what the first eight methods do: They convert their primitive arguments into bytes. (It's surprising that `writeByte()`, `writeShort()`, and `writeChar()` take `int` args rather than the corresponding primitive types. That's just how it is.) What about UTF? Recall that Java's `char` type uses Unicode encoding. So a Java string is a run of Unicode characters. [UTF](#) is a standard for converting Unicode strings into bytes. Thanks to the `writeUTF()` method, you can use a data output stream to write any of Java's eight primitives, as well as any string. This is illustrated in the following application.

The following code chains a data output stream onto a file output stream, and then it writes one of each primitive type as well as one string:

```
import java.io.*;

public class WriteWithChain
{
    public static void main(String[] args)
    {
        boolean boo = true;
        byte b = 12;
        short sh = 12345;
        char c = 'M';
        int i = -654321;
        long n = 12341234;
        float f = 15;
        double d = 1.23e88;
        String s = "Where the devil did that dragon come from?";

        try
        {
            FileOutputStream fos;
            DataOutputStream dos;

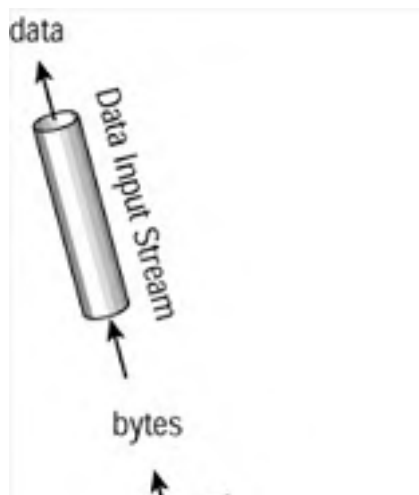
            fos = new FileOutputStream("abc");
            dos = new DataOutputStream(fos);
            dos.writeBoolean(boo);
            dos.writeByte(b);
            dos.writeShort(sh);
            dos.writeChar(c);
            dos.writeInt(i);
            dos.writeLong(n);
            dos.writeFloat(f);
            dos.writeDouble(d);
            dos.writeUTF(s);
            dos.close();
            fos.close();
        }

        catch (IOException x)
        {
            System.out.println("Caught IOException");
        }
    }
}
```

Note the two `close()` calls at the end of the try block. Every input and output stream should be closed after use. A good rule of thumb is to close chained streams in the opposite

order from their creation. Since the file output stream was constructed before the data output stream, close the data output stream first and the file output stream second.

Now you know how to write data to a file, so it is time to learn how to read data from a file. Again, you will chain a high-level stream onto a stream that communicates with a file. But this time, you will chain a data input stream onto a file input stream. [Figure 13.5](#) shows this arrangement.



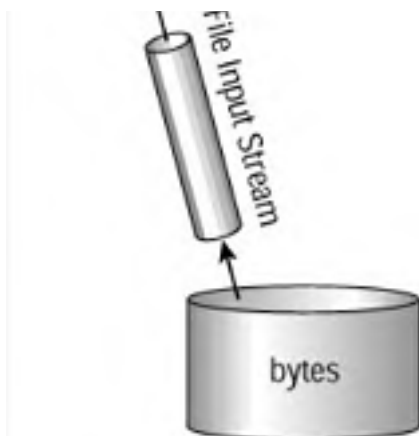


Figure 13.5: Input chaining

The `DataInputStream` class reads bytes from a lower-level stream, such as a file input stream has nine reading methods that correspond to the nine writing methods of `DataOutputStream`:

- `readBoolean()`
- `readByte()`
- `readShort()`
- `readChar()`
- `readInt()`
- `readLong()`
- `readFloat()`
- `readDouble()`
- `readUTF()`

These methods take no arguments. Their return types correspond to their names: `boolean` for `readBoolean()`, `byte` for `readByte()`, and so on. `readUTF()` returns a string. When any of these calls are made, the data input stream gets the appropriate number of bytes from its lower-level stream and assembles them to create the appropriate return value.

Now you can read the file by chaining a data input stream onto a file input stream:

```
import java.io.*;

public class ReadWithChain
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fis;
            DataInputStream dis;

            fis = new FileInputStream("abc");
            dis = new DataInputStream(fis);
            boolean boo = dis.readBoolean();
            System.out.println("Read boolean: " + boo);
            byte b = dis.readByte();
            System.out.println("Read byte: " + b);
            short sh = dis.readShort();
            System.out.println("Read short: " + sh);
            char c = dis.readChar();
            System.out.println("Read char: " + c);
            int i = dis.readInt();
            System.out.println("Read int: " + i);
            long n = dis.readLong();
            System.out.println("Read long: " + n);
            float f = dis.readFloat();
            System.out.println("Read float: " + f);
            double d = dis.readDouble();
            System.out.println("Read double: " + d);
            String s = dis.readUTF();
            System.out.println("Read string: " + s);
            dis.close();
            fis.close();
        }

        catch (IOException x)
        {
            System.out.println("Caught IOException");
        }
    }
}
```

```
}
```

This application's output is

```
Read boolean: true
Read byte: 12
Read short: 12345
Read char: M
Read int: -654321
Read long: 12341234
Read float: 15.0
Read double: 1.23E88
Read string: Where the devil did that dragon come from?
```

Obviously, this output reflects the data that was originally written to the file in the previous example. The two applications work together because the reading code reads exactly the same types, in exactly the same order, as were written by the writing code. Whenever you read from a file that was created with a data output stream, your read calls have to correspond exactly to the write calls that created the file. Otherwise, your data will be garbled beyond all recognition.

For example, suppose you mistakenly called `readLong()` instead of `readInt()`. The data input stream would grab the next eight bytes so that it could build a long. Those eight bytes would be the four-byte int (which is the next item of data in the file), and the first four bytes of the eight-byte long (which follows the int in the file).

The Data Chain Lab animated illustration demonstrates code that first writes three pieces of data to a file, and then reads them back. To run the application, type "`java io.DataChainLab`". [Figure 13.6](#) shows the display.



Figure 13.6: Data Chain Lab

In the three lines that write data, you will see pull-down choices for configuring which data type to write out. You can choose from any of the seven methods that write numerical types. You can also choose the values to be written. When you change the type being written, the corresponding reading code changes as well. This is in keeping with the rule that the type that is written must match the type that is read.

When you're ready, click the "Run" button to view the animation. If you want to run it again, perhaps with different output methods or values, first click "Reset". Then choose new methods and values, and click "Run" again.

[Figure 13.7](#) shows Data Chain Lab in progress. It has been configured to write and then read a byte, a long, and a double.

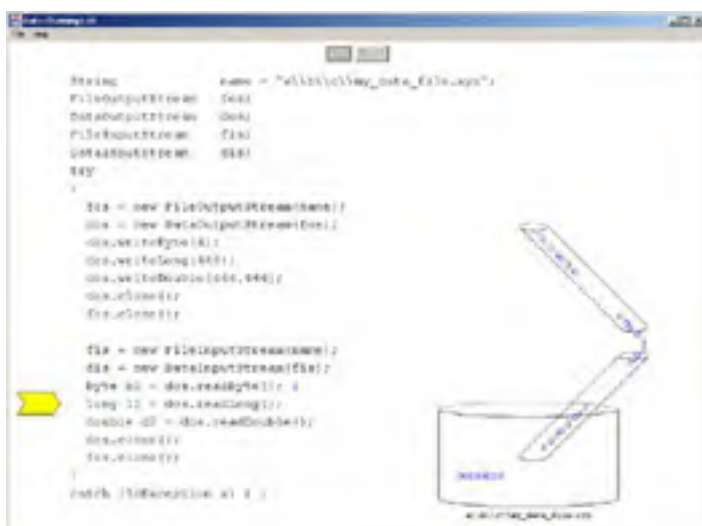


Figure 13.7: Data Chain Lab in progress: Text, writers, and readers

File output streams and file input streams are used for files that contain raw bytes. Data output streams and data input streams are used for files whose bytes represent multibyte data. There is a third kind of file, whose bytes represent text. For access to text files, Java provides classes called *readers* and *writers*.

Before presenting readers and writers, we should take a moment to explain what is meant by "text file". Recall that Java uses the modern Unicode scheme to represent text, using two bytes per character. This means there are 65,536 possible characters that can be represented. That's more than enough to represent every character of every language on the planet... until you consider Chinese, Japanese, and Korean. These non-phonetic alphabets have huge numbers of characters, enough to consume all 65,536 bit combinations. The international Unicode Consortium decides which characters of which languages will be represented by which bit combinations.

Well, that's the modern way to represent characters. It doesn't seem quite modern enough when you think about those Chinese, Japanese, and Korean symbols that get left out, but it's better than what we had before. The old way of doing things, from the invention of computers through the introduction of Unicode, was to use 8-bit characters. Every language group was on its own to decide which of the 256 possible bit combinations would represent which character. Most files created during that time used an encoding called *ASCII*, which stands for "American Standard Code for Information Interchange". ASCII encodes all the characters in American English, plus punctuation marks, into the range 0-127. The range 128-255 encodes symbols such as accented vowels, which are used in western European languages, as well as some Greek characters, line-drawing symbols, and some others. All of the characters that are represented in ASCII are represented in Unicode.

So here's the situation today: Within the JVM, characters are represented by Unicode. But in the world in general, there are millions of text files that use ASCII or other 8-bit representations. So Java needs a way to read those files and present their contents as Unicode strings. Also, Java needs a way to write ASCII files (as well as other 8-bit formats), because files can be read by non-Java programs that don't know about Unicode. Note that the problem cannot be solved by using data input and output streams that do lots of `readUTF()` and `writeUTF()` calls, because UTF is compressed Unicode, not ASCII.

Readers and writers solve the problem of translating between 16-bit characters within a JVM and 8-bit characters in text files. [Figure 13.8](#) illustrates the roles of readers and writers.

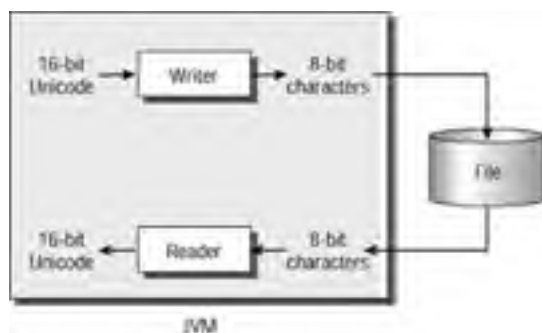


Figure 13.8: Readers and writers

`Reader` is an abstract class that reads 8-bit text and delivers Unicode chars. `Writer` is an abstract class that reads Unicode chars and delivers 8-bit text. For our purposes, the two most important subclasses of these two classes are `FileReader` and `FileWriter`, which read and write 8-bit text files.

A `FileWriter` is a lot like a `FileOutputStream`. You construct one, passing as an argument the name of the file you want to make. Then you write, and when you have finished, you close the `FileWriter`. This all must happen in a `try` block, because the code can throw `IOException` and some of its subclasses. The following code writes two lines of text to a file called `abc.txt`:

```
1. try
2. {
3.     FileWriter fw = new FileWriter("abc.txt");
4.     fw.write("Hello\n");
5.     fw.write("Goodbye\n");
6.     fw.close();
7. }
8. catch (IOException x) {}
```

Line 4 writes "Hello", followed by a newline character. Line 5 writes "Goodbye", followed by a newline character. Note that the newline is not automatic (as it is in the `System.out.println()` call, for example). If you want multiple lines of text, you have to indicate the line breaks yourself. So the following code creates an identical file:

```
1. try
2. {
3.     FileWriter fw = new FileWriter("abc.txt");
4.     fw.write("Hello\nGoodbye\n");
5.     fw.close();
6. }
7. catch (IOException x) {}
```

There are three common ways to indicate that a line has ended and a new line has begun:

- A return character (`'\r'`)
- A newline character (`'\n'`)
- A return character followed by a newline character

Which should you use? It depends on which other programs will be reading the file you create. Programs that run on Windows platforms expect a return character followed by a newline character. If you are creating files for Windows, you should do something like the following:

```
1. try
2. {
3.     FileWriter fw = new FileWriter("abc.txt");
4.     fw.write("Hello\r\nGoodbye\r\n");
5.     fw.close();
6. }
7. catch (IOException x) { }
```

If you run this code on a Windows machine and then double-click on the icon for the `abc.txt` file, Windows will open a Notepad window that displays (and lets you edit) the file. You can also open the file with any other program that reads text files, including Word.

You can read text files with the `FileReader` class, but this class is a bit limited. It is much easier to use the `LineNumberReader` class, where the `readLine()` method reads lines of text and returns strings. (This assumes that your text file has multiple lines, and that reading line by line will be useful to you. This is a safe assumption.) A call to `readLine()` reads one line from the input file. A line is a run of text, terminated by either a return character, a newline character, or a return character followed by a newline character. The line-termination characters are not part of the returned string.

A line number reader does not directly read from the input file. Rather, it is chained onto a file reader, in the same way a data input stream is chained onto a file input stream. [Figure 13.9](#) shows the relationship between a line number reader and a file reader.

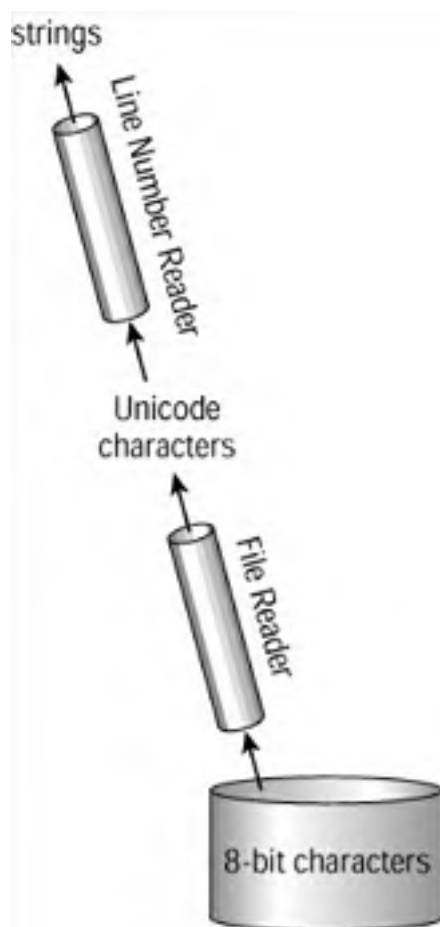


Figure 13.9: Line number reader and file reader

The following code reads and prints out the first two line of a character file:

```
try
{
    FileReader fr = new FileReader("zzz.txt");
    LineNumberReader lnr = new LineNumberReader(fr);
    System.out.println(lnr.readLine());
    System.out.println(lnr.readLine());
    lnr.close();
    fr.close();
}
catch (IOException x) { }
```

The `LineNumberReader` class keeps track of the number of lines it has read. You can retrieve the current line number by calling `getLineNumber()`.

The `readline()` method returns `null` if the end of the input file has been reached. So the following code prints out the line number of all lines in file `input.txt` that contain the word "purple":

```
1. try
2. {
3.     FileReader fr = new FileReader("input.txt");
4.     LineNumberReader lnr = new LineNumberReader(fr);
5.     String s = "";
6.     while (s != null)
7.     {
8.         s = lnr.readLine();
9.         if (s != null && s.indexOf("purple") != -1)
10.            System.out.println("Found \"purple\" at line " +
11.                               lnr.getLineNumber());
12.     }
13.     lnr.close();
14.     fr.close();
15. }
16. catch (IOException x) { }
```

The `while` loop runs as long as `s` is not `null` (that is, as long as the end of the file has not been reached). So `s` has to be initialized to anything besides `null` so that the loop will not immediately terminate. Line 9 calls `indexOf()` on the string returned by the line number reader. This method returns the position (in the string on which the method was called) of the string that is the method's argument. For example, if `s` in line 9 is "A ferocious purple dragon", the `indexOf()` call will return 12. If the argument string does not appear at all, `indexOf()` returns -1. So the condition in line 9 evaluates to `true` when the reader has not yet reached the end of the file, and the string just read contains "purple".

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. In this chapter, you learned about the following line:

```
String s = "C:my_backup\temporary\news";
```

What does the following code print out?

```
String s = "C:my_backup\temporary\news";  
System.out.println("***\n" + s + "***");
```

What is the moral of this exercise?

2. The code examples in the "[Writing and Reading Data](#)" section defined an int called `i`, a float called `f`, a double called `d`, and so on. But the long was called `n`, which breaks the pattern. You might have expected the long to be called `l`. Why do you think this was not done?
3. Write a program that creates a file containing 5,000 random doubles that are ≥ 0 and < 200 .
4. Write a program that verifies the file you created in the previous exercise. Your program should read the 5,000 doubles, making sure that each falls within the proper range. Your program should also make sure the file contains exactly 5,000 longs.
5. Look up the API documentation for the `java.io.File` class. An instance of this class contains information about an individual file. One of the methods of the class tells you the length in bytes of a file. Use this method to determine the number of bytes in the file you created in Exercise 3.

Chapter 14: Painting

We now begin a series of three chapters about visual programming. Up until now, all your applications have produced text output. In [Chapter 11, "Exceptions"](#), you learned how to provide text input via command-line arguments. Text input and output are fine, up to a point, but the mouse and the GUI provide a much richer environment for communicating a user's ideas to a program, and for communicating a program's results to a user.

Graphical user interface is usually abbreviated GUI. (Yes, it's pronounced "goeey.") The `java.awt` package contains dozens of classes that support GUI concepts like windows, colors, lines, squares, fonts, buttons, and check boxes. This chapter will show you how to display a window on your screen and paint basic shapes in it. That isn't spectacular, but this chapter will also prepare you for [Chapters 15, "Components,"](#) and 16, ["Events,"](#) where you will learn how to populate your GUIs with buttons, scrollbars, labels, and other standard controls. This chapter will end with an extended example program whose GUI combines custom painting with standard components.

Frames

A frame is a window on a computer screen, plus the "decoration" that makes it look like an independent window, plus the underlying programmatic behavior that lets you move windows around on your screen, resize them, iconify them, and so on. [Figure 14.1](#) shows a frame whose contents are gray.

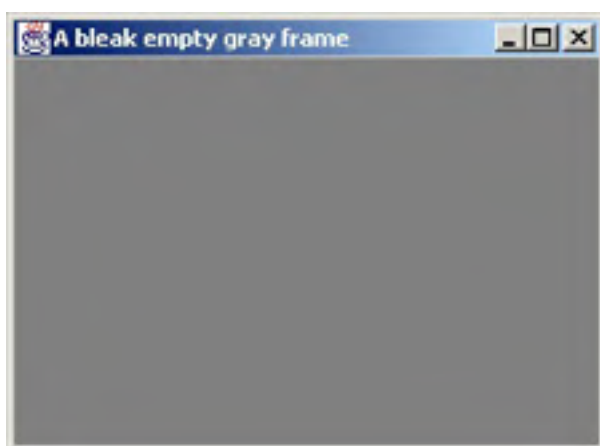


Figure 14.1: A frame with boring contents

The figure shows a Windows frame that was created by a Java program running on a Windows platform. Windows users will recognize the Minimize, Maximize, and Close buttons in the upper-right corner, as well as the decorations that give the outline its 3-D beveled appearance. On a different platform, the same program would create a frame whose controls and decorations looked slightly different, appropriate to the platform's windowing software.

So on any platform, a frame created by a Java program looks exactly like any other frame. This happens because the classes of the `java.awt` package do not directly draw components onto the screen. Instead, they instruct the underlying system's windowing software to do the work.

Note Java provides two alternative toolkits for creating GUIs. The simpler one is called AWT, which stands for Another Windowing Toolkit. The more complicated toolkit is called Swing, which doesn't stand for anything and is not discussed in this book. Swing does not use the underlying windowing software to draw components.

Here is the application that created the frame in [Figure 14.1](#):

```
1. import java.awt.*;
2.
3. public class EmptyFrame extends Frame
4. {
5.     EmptyFrame()
6.     {
7.         setTitle("A bleak empty gray frame");
8.         setBackground(new Color(128, 128, 128));
9.         setSize(300, 220);
10.    }
11.
12.    public static void main(String[] args)
13.    {
14.        EmptyFrame em = new EmptyFrame();
15.        em.setVisible(true);
16.    }
17. }
```

The application class extends `java.awt.Frame`. The superclass provides all the generic functionality of a frame. When you create a subclass of `java.awt.Frame`, you only have to provide the non-generic, application-specific behavior. In this example, the subclass does four things:

- Puts a message in the frame's title bar.

- Sets the frame's background color.
- Sets the frame's size.
- Makes the frame visible.

Line 7 sets the message in the title bar. The `setTitle()` method is inherited from the superclass; it takes a string argument.

Line 8 sets the frame's *background color*. Whenever the frame needs to be drawn on the screen, all of its pixels are set to the background color (except for the decoration pixels, of course). Then any application-specific drawing is performed. In this example, there is no application-specific drawing, so all you see in the frame's interior is uniform gray. The `setBackground()` method takes an argument of type `java.awt.Color`. We will look more deeply at this class in the [next section, "Colors."](#)

Line 9 uses the `setSize()` method to set the frame's size. The method's arguments are the desired width and height, in pixels. (A *pixel*, or *picture element*, is a dot on a display screen.) It's important to set a frame's size, because the default size is zero pixels wide by zero pixels high. So if you neglect to call `setSize()`, your frame will be too small to see.

Line 14 makes the frame visible. Before a frame executes `setVisible(true)`, it's just a lot of bytes somewhere in memory, like any other object. The first time `setVisible()` is executed, the frame establishes communication with the windowing software on the underlying system, and the windowing software draws the frame on the screen. The process is quite complicated, but it all happens automatically. You only need to remember to call both `setSize()` and `setVisible()`. Otherwise, your frame will not be seen.

Notice that the title bar message, foreground color, and size are set in the `EmptyFrame` constructor, while `setVisible()` is called in `main()`, after the frame has been constructed. The program would function the same if any of the calls on lines 7-9 were moved into `main()`. For example, the following code sets the background color and size in `main()`:

```
1. import java.awt.*;
2.
3. public class EmptyFrame extends Frame
4. {
5.     EmptyFrame()
6.     {
7.         setTitle("A bleak empty gray frame");
8.     }
9.
10.    public static void main(String[] args)
11.    {
12.        EmptyFrame em = new EmptyFrame();
13.        em.setBackground(new Color(128, 128, 128));
14.        em.setSize(300, 220);
15.        em.setVisible(true);
16.    }
17. }
```

This version produces an identical frame, but the previous version is considered better design. In the previous version, the constructor was responsible for setting the properties of the subclass instance, but it did not make the frame visible. This is clean design, because code that uses the class might want to create the object but not display it for a while. In general, constructors should set up the internal properties of an object without dictating when and how the object is to be used.

Note A frame that you create in Java does not automatically disappear when you click the "Close" button in its upper-right corner. The frame only sends an event to its listeners, using the mechanism that will be explained in [Chapter 16](#). To kill a frame, you can always type CONTROL-C in the console window where you started the program.

The code examples throughout the remainder of this book will feature frame subclasses whose constructors do everything except call `setVisible()`. Making the frame visible is the job of the code that uses the frame subclass. For us, this will always happen in `main()`, immediately after the subclass is created.

Now you know how to create a frame with boring uniform contents. Now it's time to learn how to put interesting things inside the frame. These things can be seen only if their colors are different from the frame's background color, so let's begin by looking at how Java handles colors.

Colors

Computer screens, like television screens, consist of rows and columns of tiny dots. It's difficult to see the dots with your unaided eye, but they are easy to see through a magnifying glass. The screen's electronics control each dot's color, under the direction of the computer's software or the TV's signal.

If you look closely at a pixel, you'll see that it consists of a red region, a green region, and a blue region. These are the pixel's *primary colors*.

This might contradict your childhood experiences. When you first started drawing or painting, you probably noticed that certain colors combine to make other colors. Mix red and blue to make purple. Red and yellow make orange, and blue and yellow make green. Other combinations aren't as pleasing: Red and green make an especially unpleasant brown. No doubt, someone explained to you that red, yellow, and blue are the primary colors that can be combined to make all other colors.

That isn't exactly true (there's no way to make white), but it pretty much explains the world of color. That is, until you stare too closely at a screen or start learning about computer colors. Then it seems that the primary colors are not red, yellow, and blue, but rather red, green, and blue.

Which trio is the real set of primary colors? It depends on your situation. When you mix paints, the pigments in the paints absorb certain colors from the ambient light. The remaining colors get reflected back into your eyes. Red, yellow, and blue (the primary colors of painting) are called *subtractive* primary colors because they are primary when light reflects off the absorbing pigment. Red, green, and blue (the primary colors of screens) are called *additive* primary colors because they are primary when different colors of light combine without pigment to absorb any hues. With additive primary colors, red plus green makes yellow, and red plus green plus blue makes white. You might have seen additive primaries in theaters or other venues that use colored spotlights. Where a red and green spotlight overlap, the light is yellow. Where red, green, and blue spotlights overlap, the light is white.

So when you control colors in a Java program, you have to think in terms of additive, not subtractive, primary colors. [Table 14.1](#) summarizes additive color mixing.

Table 14.1: Combining Additive Primary Colors

Primary Colors	Result
Red + green	Yellow
Red + blue	Magenta
Green + blue	Cyan
Red + green + blue	White

In Java, colors are represented by the `java.awt.Color` class. The constructor for this class has three arguments, which represent the amount of red, green, and blue that make up the color. The arguments range from a minimum of 0 through a maximum of 255. If all three arguments are 0, the color is black. If all three are 255, the color is white. As you can see from [Table 14.1](#), if red and blue are 255 while green is 0, the color is magenta. If red is 200, blue is 255, and green is 0, the color is a somewhat bluer magenta (because it contains less red).

The `Color` class has 13 predefined colors. These are public final static variables of type `Color`. (It may seem convoluted for a class to contain data of the same type as the class. That's just how it is.) The names of these variables are

- `Color.BLACK`
- `Color.WHITE`
- `Color.RED`
- `Color.GREEN`
- `Color.BLUE`
- `Color.YELLOW`
- `Color.CYAN`
- `Color.MAGENTA`
- `Color.ORANGE`
- `Color.PINK`
- `Color.LIGHT_GRAY`
- `Color.GRAY`
- `Color.DARK_GRAY`

If you want to use one of these colors, you don't have to create a new instance. For example, to set a frame's background color to orange, you can call `setBackground(Color.orange)`. If the 13 predefined colors don't give you what you want, you need to construct your own. You might find that `Color.orange` isn't intense enough. Its green level is 200, which tends to wash out the brilliance of the red. A nice intense orange is created by calling `new Color(255, 200, 0)`. You can make this the background color of a frame by calling `setBackground(new Color(255, 200, 0))`.

Looking at colors is better than reading about them. The Color Lab program lets you practice mixing primaries, and it also shows you all 13 predefined colors. To run the program, type `java visual.ColorLab`. [Figure 14.2](#) shows the initial display.

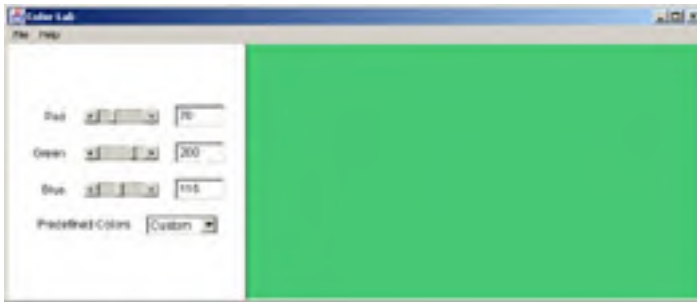


Figure 14.2: Color Lab

The control panel contains three scrollbars, with a text field showing the value of each scrollbar, as well as a pull-down menu. When you choose Custom from the pull-down menu, as shown in [Figure 14.2](#), the scrollbars are enabled. You can set them to any value from 0 through 255, and the display area to the right of the control panel will show the color you've specified. You can also type numbers into the text fields. Press Enter to make your entry take effect.

In addition to Custom, the pull-down menu lets you choose any of the 13 predefined colors of the `Color` class. When one of these is selected, the scrollbars and text fields are disabled. They display the red/green/blue levels of the selected color, but you can't use them for input. [Figure 14.3](#) shows Color Lab displaying a predefined color.



Figure 14.3: Color Lab with a predefined color

Notice that the sliders have no bubbles, and the numbers in the text fields are gray, indicating that those components are not enabled to receive user input.

Set the Color Lab inputs to display yellow. You can do this by selecting YELLOW, or by selecting Custom and manipulating the scrollbars. Now look at the yellow area of the screen through a magnifying glass. Observe the separate red and green areas of the individual pixels. Have a friend hold the magnifying glass steady, and move slowly backwards until the red and green seem to coalesce into yellow. How far from the screen are you when this begins to happen? You are invited to e-mail the distance to us at groundupjava@sgsware.com. We will compile the statistics and publish them on our website.

Now you know how to use Java's predefined colors, and how to construct a custom color when you need one. Now let's move on to using colors to draw shapes inside a frame.

Painting

There are many reasons why a frame's contents may need to be redrawn:

- The frame has become visible for the first time.
- The contents have changed due to user input (as when you moved a scrollbar in Color Lab).
- The frame has deiconified.
- The frame has moved to the front of the desktop, after having been covered or partially covered by another frame.

When any of these occur, the underlying windowing software notifies the frame, and a call is made to the frame's `paint()` method. The great thing about this arrangement is that you never have to detect these changes to the frame. The environment takes care of all that for you. All you have to do is subclass `Frame` and provide a `paint()` method that draws the frame's interior. In other words, you have to think about *what* to paint, but you don't have to think about *when* to paint.

When the environment decides that a frame needs painting, the frame's interior is cleared to its background color. By default, the background color is white. But as you saw in the [previous section](#), you can call `setBackground()` to set any background color you like. After that, the environment calls the frame's `paint()` method. The `paint()` version inherited from `Frame` does nothing at all. You are about to learn how to override `paint()` so that it does interesting things.

The argument of `paint()` is an instance of `java.awt.Graphics`. You might hear people call this object a *graphics context*, but it's more correct to call it a *graphics object*, and that is the name we will use. The graphics object is like an artist with a paintbrush, ready to paint the interior of a frame. It isn't a very talented artist (it only knows how to draw a few shapes), but it's very accurate. And as you'll see, it has excellent penmanship. You never have to construct an instance of `Graphics`; that's done for you by the environment. You just have to tell it what to paint.

An artist at work dips his brush in paint, brushes the paint onto paper, dips, brushes, and so on. The color that goes on the paper is, of course, the last color that the brush was dipped into. A graphics object works the same way. It has a method called `setColor()`, whose argument is a `Color`. It also has methods that draw shapes, including lines, rectangles, circles, and text messages. The shapes appear in the color that was the argument of the most recent `setColor()` call. So you can see that calling `setColor()` is like dipping your paintbrush into new paint. Another way to think of it is this: When you call `setColor()`, you set the color of all shapes to be drawn until the next `setColor()` call.

Now let's take a look at the different shapes that a graphics object can paint.

Drawing and Filling with a Graphics Object

The shapes that you can draw with the `Graphics` class include the following:

- Lines
- Squares and rectangles
- Circles and ovals

There are also methods that fill the interior of a square, rectangle, circle, or oval. All drawing happens in the color of the most recent `setColor()` method, as you saw in the [previous section](#). The methods have varying arguments that specify the size and location of the shape. All arrays are in units of pixels, not inches or millimeters. Horizontal positions are always called *x*, and are measured from the left edge of the frame. Vertical positions are called *y*, and are measured from the top of the frame. The location of a point is denoted by (*x*, *y*), as shown in [Figure 14.4](#).



Figure 14.4: Pixel coordinates

The point at (0, 0) is called the *origin*. A frame's origin is its top-left pixel. Note that *x* increases from left to right, and *y* increases

from top to bottom. This is different from the Cartesian coordinates that you may have learned about in school, where y increases upward. The y-increases-downward scheme is standard in graphical programming, and it often causes confusion until people get used to it. It probably got its start in word-processing software, where line numbers increase from the top to the bottom of a document. Whatever its derivation might be, the scheme is here to stay.

To draw a line from (x0, y0) to (x1, y1), call the following on your graphics object:

```
drawLine(x0, y0, x1, y1);
```

The following code displays a frame with a black line on a white background:

```
1. import java.awt.*;
2.
3. public class BlackLineOnWhite extends Frame
4. {
5.     BlackLineOnWhite()
6.     {
7.         setSize(150, 180);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        g.setColor(Color.black);
13.        g.drawLine(60, 115, 120, 70);
14.    }
15.
16.    public static void main(String[] args)
17.    {
18.        BlackLineOnWhite blonw = new BlackLineOnWhite();
19.        blonw.setVisible(true);
20.    }
21. }
```

Figure 14.5 shows the frame.



Figure 14.5: A black line on a white background

The constructor just sets the frame's size. There's no need to set the background color explicitly, since you want the white default. Line 12 actually isn't required, because when `paint()` is called, the graphics object is set up to draw in black automatically.

To draw a rectangle, call the `drawRect()` method. Its four arguments are the x, y, width, and height of the rectangle, where (x, y) is the location of the rectangle's upper-left corner. The following code draws a blue rectangle that is 100 pixels wide by 35 pixels high, with its upper-left corner at (25, 50):

```
1. import java.awt.*;
2.
3. public class BlueRect extends Frame
4. {
5.     BlueRect ()
6.     {
7.         setSize(150, 180);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        g.setColor(Color.blue);
13.        g.drawRect(25, 50, 100, 35);
14.    }
15.
16.    public static void main(String[] args)
17.    {
18.        BlueRect br = new BlueRect();
19.        br.setVisible(true);
20.    }
21. }
```

Figure 14.6 shows the frame.



Figure 14.6: A rectangle

There is no separate method for drawing a square. You just call `drawRect()` with equal values for the width and height.

To draw an oval, you call `drawOval()` and specify the oval's *bounding box*. A bounding box is the smallest rectangle that encloses the oval. [Figure 14.7](#) shows several ovals and their bounding boxes.

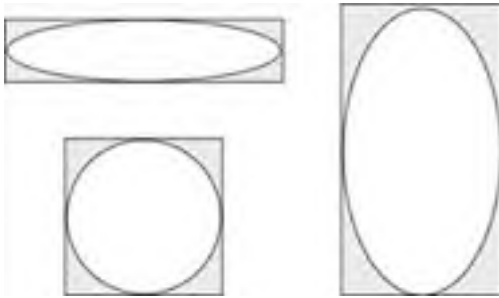


Figure 14.7: Ovals and bounding boxes

Notice that one of the shapes in [Figure 14.7](#) looks like a circle, not an oval. Actually, a circle is a kind of oval whose bounding box is a square.

The following code draws three ovals (shown in [Figure 14.8](#)), one of which is a circle:

```
1. import java.awt.*;
2.
3. public class ThreeOvals extends Frame
4. {
5.     ThreeOvals ()
6.     {
7.         setSize(150, 220);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        g.setColor(Color.black);
13.        g.drawOval(20, 40, 100, 35);
14.        g.drawOval(20, 85, 50, 60);
15.        g.drawOval(90, 105, 25, 25);
16.    }
17.
18.    public static void main(String[] args)
19.    {
20.        ThreeOvals throv = new ThreeOvals ();
21.        throv.setVisible(true);
22.    }
23. }
```

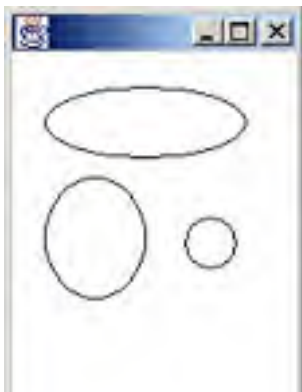




Figure 14.8: Three ovals

In the `drawOval()` calls, the arguments are the `x`, `y`, width, and height of the bounding box. Notice that in line 15, which draws the circle, the width and height are the same.

The `fillRect()` method draws a rectangle and fills its interior. The `fillOval()` method draws an oval and fills its interior. The following code displays two filled ovals and a filled rectangle:

```
1. import java.awt.*;
2.
3. public class Filled extends Frame
4. {
5.     Filled ()
6.     {
7.         setSize(200, 150);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        g.setColor(Color.black);
13.        g.fillOval(20, 50, 50, 20);
14.        g.fillRect(90, 40, 20, 70);
15.        g.fillOval(130, 50, 50, 20);
16.    }
17.
18.    public static void main(String[] args)
19.    {
20.        Filled f = new Filled();
21.        f.setVisible(true);
22.    }
23. }
```

The result is shown in [Figure 14.9](#).



Figure 14.9: Filled rectangle and ovals

Our last example in this section draws a filled oval that is centered in its frame. The oval is half as high and half as wide as the frame. The code uses the frame's `getSize()` method, which is inherited from one of the superclasses of `java.awt.Frame`. This method returns an instance of `Dimension`, which is a tiny class with two public ints called `width` and `height`:

```
1. import java.awt.*;
2.
3. public class CenteredOval extends Frame
4. {
5.     CenteredOval ()
6.     {
7.         setSize(200, 150);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        g.setColor(Color.black);
13.        Dimension size = getSize();
14.        g.fillOval(size.width/4, size.height/4,
15.                  size.width/2, size.height/2);
16.    }
17.
18.    public static void main(String[] args)
19.    {
20.        CenteredOval cenOv = new CenteredOval ();
21.        cenOv.setVisible(true);
22.    }
23. }
```

[Figure 14.10](#) shows the frame in its original size.



Figure 14.10: Original CenteredOval

If you replace line 7 with `setSize(400, 300);`, you get [Figure 14.11](#).

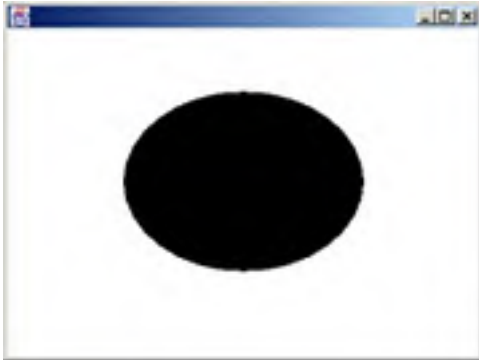


Figure 14.11: Resized CenteredOval

No matter what size is assigned to the frame in line 7, the `paint()` method always draws an oval with the correct proportions.

Now you know how to use a graphics object to do the following:

- Draw lines.
- Draw rectangles, including squares.
- Fill rectangles, including squares.
- Draw ovals, including circles.
- Fill ovals, including circles.

In the [next section](#), you'll learn how to draw text.

Text and Fonts

To draw text in a frame, call the graphics object's `drawString()` method. The method's arguments are the string to be drawn followed by its *x* and *y* position. The *x* position is the leftmost pixel of the first character in the string. The *y* position is the location of the baseline. The *baseline* of a string is the bottom of all characters except *g*, *j*, *p*, *q*, and *y*, which descend below the baseline. When you write on lined paper, the lines are baselines. [Figure 14.12](#) shows a string containing two characters that descend below the baseline.

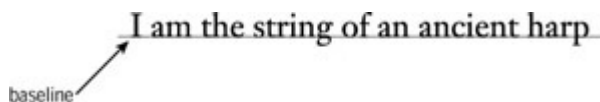


Figure 14.12: The baseline

The following code draws a string that contains six descending characters. It also draws the baseline in light gray:

```
1. import java.awt.*;
2.
3. public class FontAndBaseline extends Frame
4. {
5.     FontAndBaseline()
6.     {
7.         setSize(200, 150);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        int x = 25;
13.        int yBaseline = 100;
14.        g.setColor(Color.lightGray);
15.        g.drawLine(x, yBaseline, 125, yBaseline);
16.        g.setColor(Color.black);
17.        g.drawString("just a gaping quay", x, yBaseline);
18.    }
19.
20.
21.    public static void main(String[] args)
22.    {
23.        FontAndBaseline fabl = new FontAndBaseline();
24.        fabl.setVisible(true);
25.    }
26. }
```

The `drawString()` call is on line 17. [Figure 14.13](#) shows the frame.



Figure 14.13: Text and baseline in a frame

You can use the graphics object's `setFont()` method to control the font in which text is displayed. Calling `setFont()` before calling `drawString()` is a bit like calling `setColor()` before drawing or filling a shape. The `setFont()` call determines the font of all subsequent `drawString()` calls, until the next `setFont()` call.

A font has three properties:

- Style
- Size
- Family

The style can be either plain, **bold**, *italic*, or **bold-italic**. The size is in pixel units.

The font families that are available to you vary from one machine to the next, but there are three that you can always count on:

- Monospaced
- Serif
- Sans Serif

In Monospaced font, all characters are spaced equally. The spacing is determined by the font's size. It's difficult to read a dense block of text in Monospaced font, but it's ideal for source code. All the code listings in this book appear in Monospaced font.

Serif is a *variable-width font*, which means that characters have different widths. For example, *i* is narrower than *m*. This book is printed in a variable-width font. Notice how *iiiiiiii* is much narrower than *mmmmmmmm*, even though both are 10 letters long. Variable-width fonts are designed for easy reading of dense text, such as you see in a book or newspaper. This font uses *serifs*, which are small decorations on the tips of letters. Serifs improve readability in medium to large fonts, but are annoying in small fonts.

Sans Serif is a variable-width font that does not use serifs. ("Sans" is French for "without".) This font works best when the font is small enough that serifs would interfere with readability.

To use a font in Java, you must first construct an instance of the `java.awt.Font` class. The constructor takes three arguments: the family, the style, and the size. The family is a string; the style and size are ints. For these three families, the strings are `Monospaced`, `Serif`, and `SansSerif`. For plain, bold, and italic, the `Font` class provides public final static ints named `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. For bold-italic style, use `Font.BOLD + Font.ITALIC`.

After you construct an instance of `Font`, you can pass it into the `setFont()` method of a graphics object. The following code sets a 36-point bold-italic Serif font:

```
Font f = new Font("Serif", Font.PLAIN+Font.BOLD, 36);  
g.setFont(f);
```

When these lines are inserted between lines 16 and 17 in the previous example (that is, just before the `drawString()` call), the result is as shown in [Figure 14.14](#).



Figure 14.14: Text in a frame

Your computer probably has dozens of fonts in addition to the three standard ones. Many of them may be more interesting and playful than Monospaced, Serif, and Sans Serif. The more stylized fonts tend to be appropriate in more limited situations.

When you use a non-standard font in a Java application, be aware that you're taking a risk. It's possible that the font won't be available on all computers that will be running your application. When this happens, all characters will appear on the screen as small empty rectangles.

The `GraphicsEnvironment` class contains information about a computer's graphics system, including the names of all available fonts. The class has a static method called `getLocalGraphicsEnvironment()`, which returns an instance of the class with all the data fields set to reflect the capabilities of the underlying computer. Another call is `getAvailableFontFamilyNames()`, which is not static. It returns the font families as an array of strings. So, to retrieve the array, you can do something like the following:

```
GraphicsEnvironment grenv =  
    GraphicsEnvironment.getLocalGraphicsEnvironment();  
String[] names = grenv.getAvailableFontFamilyNames();
```

The `Font Lab` program lets you see all the fonts that are available on your computer. To run it, type `java visual.FontLab`. You will see the GUI shown in [Figure 14.15](#).



Figure 14.15: Font Lab

The pull-down menu in `Font Lab`'s control panel lets you choose from among all of the fonts on your machine, as detected by `getAvailableFontFamilyNames()`. You can change the family, style, and size. Some families do not support all styles. Some have only plain and italic, and others have only plain. [Figure 14.16](#) shows one of the more exotic fonts.

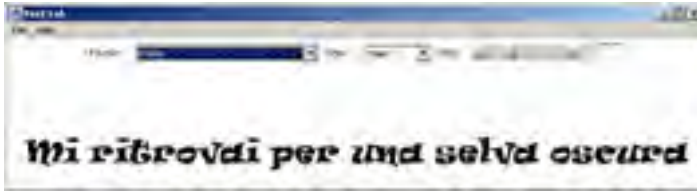


Figure 14.16: Font Lab with an exotic font

Play around with the various fonts. Find one that seems serious and one that seems playful. How do their shapes differ? Select 16-point plain Serif. Reduce the size one point at a time until the font becomes hard to read. Do the same for SansSerif. You will probably find that SansSerif remains readable down to a slightly smaller size than Serif.

Team LiB

← PREVIOUS NEXT →

Frame Lab

The Frame Lab animated illustration lets you practice setting colors, drawing shapes, setting fonts, and drawing text. To run the program, type `java visual.FrameLab`. The initial display is shown in [Figure 14.17](#).



Figure 14.17: Initial Frame Lab display

The animated illustration simulates a subclass of `Frame`, with a `paint()` method that you can configure. You can enter any class name you like in the first text field. (Notice that the lines that declare and call the constructor change as you change the name.) If you want your constructor to set the background color, make sure the check box in the `setBackground` line of the constructor is checked, and select a color from the pull-down menu. The constructor sets the frame's size to 450 by 450, but you can enter any number you want.

The body of the `paint()` method consists of ten lines, each controlled by its own pull-down menu that lets you select a method to call on the graphics object. You can set the color or the font, or you can call any of the drawing methods that were presented in this chapter. You can also select a comment (`*****`), which indicates that you don't want the line to do anything. No matter what you select, the line will present you with controls for entering the arguments of the method you've chosen.

When you're ready to simulate execution of the code you've set up, click either `Run` or `Run Lightspeed` to view either an animation or an instant result. If you want to run again, click `Reset`, adjust the controls, and again click either `Run` or `Run Lightspeed`. [Figure 14.18](#) shows one possible Frame Lab configuration.



Figure 14.18: Frame Lab with custom configuration

The resulting frame is shown in [Figure 14.19](#).



Figure 14.19: The result of [Figure 14.18](#)

Try configuring Frame Lab to paint the following:

- A line of text, in any font you like, with the baseline visible.
- A line of text centered in a filled oval.
- Three concentric circles.

Now draw anything you like. If you create any interesting results that you would like to share, please email them to us at groundupjava@sgsware.com. In the next edition of this book, Frame Lab will include a gallery of the best pictures submitted, along with the artists' names.

Take a look at the `main()` method in Frame Lab. So far in this chapter, all your `main()` methods have been two lines long, like this:

```
public static void main(String[] args)
{
    FontAndBaseline fabl = new FontAndBaseline();
    fabl.setVisible(true);
}
```

In Frame Lab, vertical space is a valuable commodity. That's why the open curly brackets appear at the ends of lines, rather than on their own lines. Frame Lab's `main()` is only one line long:

```
(new FancyFrame()).setVisible(true);
```

This is just a shorter equivalent of the following:

```
FancyFrame ff = new FancyFrame();
ff.setVisible(true);
```

In the single-line version, there is no reference to the instance of `FancyFrame` that gets constructed. If you want to make another call on the instance after `setVisible()`, you're out of luck. The instance is *anonymous*. "Anonymous" means "without a name," and a reference is like an object's name. The single-line version of the constructor is considered better style, because there is no need for the reference except in the `setVisible()` call. For the rest of this book, the `Frame` subclass instances will be anonymous.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. The first code example in this chapter used the following code to set a frame's background color:

```
setBackground(new Color(128, 128, 128));
```

Describe the color that this line creates.

2. Run Color Lab, and adjust the scrollbars so that the displayed color matches something you can see (a piece of clothing you're wearing, or something on your desk, or anything else you like). Now write an application that displays a frame whose interior is the color you've chosen.
3. One of the code examples in this chapter used the `getSize()` method, which `Frame` inherits from one of its superclasses. Use the API to find out which superclass implements the method.
4. Write a program that draws a five-pointed star. Your frame should be 400 x 400 pixels. The coordinates of the star's points are (200, 375), (97, 58), (366, 254), (34, 254), and (303, 58). The easy way is to write a `paint()` method that calls `drawLine()` five times. But that approach isn't ideal, because you have to type each `x` and each `y` twice. (Each point is the end of two lines, so it appears in two `drawLine()` calls.) Typing data, code, or anything else more than once is considered bad style. If one of the copies has a typo and doesn't match the original precisely, your program won't function correctly. To avoid duplication of data, your program should have two `int` arrays, defined as follows:

```
int[] xs = {200, 97, 366, 34, 303};  
int[] ys = {375, 58, 254, 254, 58};
```

Your `paint()` method should have a loop that accesses these arrays. `drawLine(...)` should appear only once in your code, inside the loop.

5. Write a program that lists all the font families that are available on your computer.

Chapter 15: Components

A *component* is a GUI device that presents user input to programs and displays program information to users. Standard GUI components include buttons, text fields, scrollbars, and menus. Java's `awt` package provides a rich suite of components, all of which are reasonably easy to use. A program's GUI can combine custom drawing, which you learned about in the [previous chapter](#), with standard components, which you will learn about here and in the [next chapter](#).

You have probably heard the expression "[look and feel](#)." For example, several years ago there was a major lawsuit between Apple and Microsoft; one company alleged that the other had plagiarized their look and feel. A program's *look and feel* consists of its appearance (look) plus its responses to user input (feel). This chapter will focus on look; feel will be covered in [Chapter 16](#), "[Events](#)."

A Survey of Components

In this section, you will learn about some of the most useful components of the `awt` package:

- Buttons
- Checkboxes
- Choices
- Labels
- Menus
- Text fields
- Text areas
- Scrollbars

You have probably encountered all of these component types in the course of using your computer. As a reminder, [Figure 15.1](#) shows one of each component type listed.



Figure 15.1: A component sampler

Now let's jump in and learn about each of these components.

Buttons

Buttons are perhaps the most familiar of all component types. We are so accustomed to them that we take for granted statements like, "Click on the OK button on your screen to confirm your purchase." Of course, there isn't really a button on the screen; it's just a picture of a button. "Press the OK button" really means, "Move your mouse until the arrow on the screen is over the picture of the button. Then press and release the button on your mouse."

[Figure 15.2](#) shows a button in a frame.





Figure 15.2: A button in a frame

In Java, buttons are represented by the `java.awt.Button` class. Here is the code that created [Figure 15.2](#):

```
1. import java.awt.*;
2.
3. class ShowButton
4. {
5.     public static void main(String[] args)
6.     {
7.         Frame f = new Frame("Simple Button");
8.         LayoutManager lom = new FlowLayout();
9.         f.setLayout(lom);
10.        Button btn = new Button("Hello");
11.        f.add(btn);
12.        f.setSize(200, 200);
13.        f.setVisible(true);
14.    }
15. }
```

Line 10 illustrates the most common `Button` constructor, which takes a string argument. The string appears as the button's text label.

Something in this code is conspicuously absent, and something else in conspicuously mysterious. Look at [Figure 15.2](#). The button is a reasonable size. It's big enough to encompass its text, and not much bigger. It's near the top of the frame, and it's horizontally centered. Conspicuously absent is the code that sets the button's size and location.

The mysterious code is lines 8 and 9, which construct and use an instance of `FlowLayout`. The `FlowLayout` class is a kind of *layout manager*. Layout managers are responsible for setting the location and size of components. We will visit them in detail in the second half of this chapter; you will find them much easier to understand after you know about components. For now, be aware that the button's reasonable size and location were set by the layout manager. The call to `add()` on line 11 puts the button in the frame. The method uses the layout manager to work out the details.

The example code is not very object-oriented. Here is a version that extends `Frame`:

```
1. import java.awt.*;
2.
3. class BtnInAFrame extends Frame
4. {
5.     public BtnInAFrame()
6.     {
7.         setLayout(new FlowLayout());
8.         Button btn = new Button("Hello");
9.         add(btn);
10.        setSize(200, 200);
11.    }
12.
13.    public static void main(String[] args)
14.    {
15.        (new BtnInAFrame()).setVisible(true);
16.    }
17. }
```

The GUI that this code produces is identical to the previous example. Notice that construction and use of the layout manager have now been combined into a single line (line 7) so as to be less obtrusive.

In the [previous chapter](#), you learned about fonts and colors. The `Button` class has three methods that let you control the font and color of a button:

- `setFont(Font f)`
- `setForeground(Color c)`
- `setBackground(Color c)`

Actually, `Button` inherits these methods from its superclass, `java.awt.Component`. All the component classes you will learn about in this chapter extend `java.awt.Component`, so they all implement these three methods.

`setFont()` sets the font of any text that the component displays. `setForeground()` sets the color of the component's text, and `setBackground()` sets the component's background color. The following code displays a button with a large yellow serif font on a blue background:

```
import java.awt.*;

class FancyButtonInFrame extends Frame
{
    public FancyButtonInFrame()
    {
        LayoutManager lom = new FlowLayout();
        setLayout(lom);
        Button btn = new Button("Hello");
        Font font = new Font("Serif", Font.ITALIC, 36);
        btn.setFont(font);
        btn.setForeground(Color.yellow);
        btn.setBackground(Color.blue);
        add(btn);
        setSize(200, 200);
    }
}
```

```
}  
  
public static void main(String[] args)  
{  
    FancyButtonInFrame b = new FancyButtonInFrame();  
    b.setVisible(true);  
}  
}
```

Figure 15.3 shows the button in the frame. The black-and-white screen shot does not do justice to the colors, but you can clearly see the enlarged italic font.



Figure 15.3: A fancy button

Notice that the button is still large enough to encompass its text, even though the text is now considerably larger. We have the mysterious layout manager to thank for that.

Buttons are for clicking. If you run any of the code in this section, you will find that the buttons do the right thing when you click on them: They look like they're indented into the screen until you release the main mouse button. However, nothing happens within the program. This should be expected, since there is no code in any of the programs that appears to deal with listening for button input. Listening for input is a function of feel, not look, so it will be presented in the [next chapter](#).

Checkboxes

A checkbox is a little box that can be either checked or not checked. The checked/not checked state changes whenever the user clicks on the component. The two most useful constructors are

- `Checkbox(String s)`
- `Checkbox(String s, boolean state)`

The string is the checkbox's text. The boolean in the second form is the checkbox's initial state. The following code builds and displays a simple unchecked checkbox:

```
import java.awt.*;  
  
class CboxInnaFrame extends Frame  
{  
    public CboxInnaFrame ()  
    {  
        setLayout(new FlowLayout());  
        Checkbox cbox = new Checkbox("Check Me");  
        add(cbox);  
        setSize(200, 200);  
    }  
  
    public static void main(String[] args)  
    {  
        (new CboxInnaFrame ()).setVisible(true);  
    }  
}
```

The result is shown in [Figure 15.4](#).





Figure 15.4: A simple checkbox

The figure just shows the initial state of the GUI. If someone clicks on the box, it will be checked.

The following code displays a checkbox that is checked if the application was invoked with "yes" as its first command-line argument.

```
import java.awt.*;

class CheckedCbox extends Frame
{
    public CheckedCbox(boolean b)
    {
        setLayout(new FlowLayout());
        Checkbox cbox = new Checkbox("Check Me", b);
        add(cbox);
        setSize(200, 200);
    }

    public static void main(String[] args)
    {
        boolean state = false;
        if (args.length > 0 && args[0].equals("yes"))
            state = true;
        (new CheckedCbox(state)).setVisible(true);
    }
}
```

Figure 15.5 shows the result when the application is invoked by typing `java CheckedCbox yes`.



Figure 15.5: A checked checkbox

Now it's time to display several components together. The following code creates three checkboxes and a button:

```
1. import java.awt.*;
2.
3. class Boats extends Frame
4. {
5.     Checkbox[]   cboxes;
6.     Button       btn;
7.     String[]     sizes = { "small", "medum", "large" };
8.
9.     Boats()
10.    {
11.        setLayout(new FlowLayout());
12.
13.        cboxes = new Checkbox[sizes.length];
14.        for (int i=0; i<sizes.length; i++)
15.        {
16.            String s = "a " + sizes[i] + " boat";
17.            cboxes[i] = new Checkbox(s);
18.            add(cboxes[i]);
19.        }
20.        btn = new Button("Add to shopping cart");
21.        add(btn);
22.
23.        setSize(600, 200);
24.    }
25.
26.    public static void main(String[] args)
27.    {
28.        new Boats().setVisible(true);
29.    }
30. }
```

This application is slightly longer than any of our previous GUI code examples. It is long enough to warrant some structure. Note that the three checkboxes, which could have been constructed one by one, are constructed in a loop. Line 16 generates the text for each checkbox, based on the appropriate string from the `sizes` array on line 7. A nice benefit of this structure is the visual isolation of the literal strings. They are easy to find, up near the top of the code listing.

Did you notice that "medium" was misspelled? Would you have noticed so easily if the literal strings were in the middle of the code? Imagine the difficulty in correcting a spelling error if the strings were scattered over a 300-line constructor. When you misspell a keyword (like "for" or "new"), the compiler tells you the line number where the error occurs. But when you misspell a literal string, the compiler can't help you. You have to go hunting for the string.

Another benefit of this program's structure is the ease with which it can be modified. If you want to add or delete some sizes, changing the array on line 7 is the only change you need to make. Exercises 2 and 3 show this in action.

Figure 15.6 shows the example program's GUI. (The spelling error has been fixed.)



Figure 15.6: Three checkboxes and a button

In the figure, the user has checked both "a small boat" and "a large boat". This is suspicious. A GUI should capture the user's precise intention. Moreover, a well-designed GUI should make it impossible for a user to enter invalid data. Figure 15.6 gives the impression that the user is supposed to check only one of the three checkboxes. If the checkboxes represent mutually exclusive alternatives, the GUI should be changed to discourage (or, better yet, prevent) selection of more than one boat size.

There are two ways to change the GUI:

- Insert text that tells the user to check only one box.
- Insert code that automatically unchecks a box whenever the user makes a new selection.

The first option puts all the responsibility on the user. The GUI still permits invalid input, and the user gets all the blame when something goes wrong. This approach is unforgivable. It's also distressingly common: Every Web user has experienced an extreme version of it. Think of the last time you typed your credit card number or phone number into a Web page, only to be told that you should have (or should not have) used spaces or hyphens. Then you have to wait for the page to reload, you have to reenter your credit card or phone number, and if the page designer was especially inept, you have to reenter your name and address as well.

The second option makes it impossible for any user to select more than one option. The result is a GUI that is free from blame. This is the approach we will take.

Java's checkboxes can act as radio buttons. A *radio button* is a member of a group, only one of which can be selected at any time. The term comes from the station-selection buttons on a car radio. To give radio-button behavior to a group of checkboxes, you first create an instance of the class `java.awt.CheckboxGroup`:

```
CheckboxGroup cbg = new CheckboxGroup();
```

When you construct your checkboxes, use one of the following constructors:

```
Checkbox(String s, boolean state,  
         CheckboxGroup cbg)
```

or

```
Checkbox(String s, CheckboxGroup cbg,  
         boolean state)
```

Here is the previous example, rewritten to use a checkbox group:

```
1. import java.awt.*;  
2.  
3. class RadioBoats extends Frame  
4. {  
5.     Checkbox[]    cboxes;  
6.     Button        btn;  
7.     String[]     sizes = { "small", "medium", "large" };  
8.  
9.     RadioBoats()  
10.    {  
11.        setLayout(new FlowLayout());  
12.  
13.        cboxes = new Checkbox[sizes.length];  
14.        CheckboxGroup cbg = new CheckboxGroup();  
15.        for (int i=0; i<sizes.length; i++)  
16.        {  
17.            String s = "a " + sizes[i] + " boat";  
18.            boolean state = (i == 0);  
19.            cboxes[i] = new Checkbox(s, state, cbg);  
20.            add(cboxes[i]);  
21.        }  
22.        btn = new Button("Add to shopping cart");
```

```
22.     btn = new JButton( "Add to shopping cart" );
23.     add(btn);
24.
25.     setSize(600, 200);
26. }
27.
28. public static void main(String[] args)
29. {
30.     new RadioBoats().setVisible(true);
31. }
32. }
```

Line 18 creates a boolean whose value is `true` in the first pass through the loop. Thus, the first checkbox is checked and the rest are not. [Figure 15.7](#) shows the GUI after the "a large boat" box has been selected.



Figure 15.7: Checkboxes as radio buttons

Notice the appearance of the checkboxes. There are no check marks, and there are no boxes. The circular buttons are a standard visual cue that the components have radio behavior. This cue is standard not just in Java, but in all current windowing toolkits.

When you create a GUI that has multiple checkboxes, ask yourself if the checkboxes can be selected independently, or if only one should be selected at any moment. If they are independent, use plain checkboxes. If they are exclusive, give them radio behavior by creating a checkbox group for them.

Choices

Suppose you want to create a GUI for specifying a font. Suppose also that you want your users to choose one of the three standard font families, and also to choose a size from among a small set of options. You might use the following code:

```
import java.awt.*;

class ChooseFontByRadios extends Frame
{
    String[] families = {"Monospaced", "Serif", "SansSerif"};
    int[] sizes = {16, 24, 32, 64};

    public ChooseFontByRadios()
    {
        setLayout(new FlowLayout());

        CheckboxGroup familyCBG = new CheckboxGroup();
        for (int i=0; i<families.length; i++)
            add (new Checkbox(families[i], (i==0), familyCBG));

        CheckboxGroup sizeCBG = new CheckboxGroup();
        for (int i=0; i<sizes.length; i++)
            add (new Checkbox(""+sizes[i], (i==0), sizeCBG));

        setSize(500, 200);
    }

    public static void main(String[] args)
    {
        (new ChooseFontByRadios()).setVisible(true);
    }
}
```

This code creates two checkbox groups. The result, as you can see in [Figure 15.8](#), is less than brilliant.



Figure 15.8: Multiple checkbox groups

The problem with the GUI is that there are no visual cues to tell you that there are two independent groups of checkboxes. This brings us to an important principle of GUI design: Components that are *functionally* related should also be *visually* related. To create a sensation of visual relationship among a group of components, you need to do two things:

- Place the components near one another.
- Isolate them from other nearby components.

In [Figure 5.8](#), the components that control the font family are certainly near one another, but they are not isolated from the components that control size.

The `java.awt.Choice` component class offers an alternative to groups of checkboxes. A *choice* is a single component that lets the user make a one-of-many selection. [Figure 15.9](#) shows a simple choice.



Figure 15.9: A choice

Choices are like pull-down menus. When you click on the component, the entire set of options is displayed, as shown in [Figure 15.10](#).



Figure 15.10: An expanded choice

Here is the code that created the GUIs in [Figures 15.9](#) and 15.10:

```
1. import java.awt.*;
2.
3. class SimpleChoice extends Frame
4. {
5.     String[] families = {"Monospaced", "Serif",
6.                         "SansSerif"};
7.
8.     public SimpleChoice()
9.     {
10.        setLayout(new FlowLayout());
11.        Choice c = new Choice();
12.        for (int i=0; i<families.length; i++)
13.            c.add(families[i]);
14.        add(c);
15.        setSize(200, 200);
16.    }
17.
18.    public static void main(String[] args)
19.    {
20.        (new SimpleChoice()).setVisible(true);
21.    }
22. }
```

The choice is created in line 11. Notice that there are no constructor arguments. Line 13 calls the choice's `add()` method to add more options; the method's argument is a string. After the choice is constructed and populated, it is added to the GUI at line 14. Notice that line 14 uses the `add()` method of the frame to add the choice component to the frame. This is a different method from the `add()` in line 13, which adds options to the choice.

The following application uses choices to support selection of a font family and size:

```
1. import java.awt.*;
2.
3. class FontChoice extends Frame
4. {
5.
6.     String[]  families = {"Monospaced", "Serif",
7.                          "SansSerif"};
8.     int[]     sizes    = {16, 24, 32, 64};
9.
10.    public FontChoice()
11.    {
12.        setLayout(new FlowLayout());
13.        Choice c = new Choice();
14.        for (int i=0; i<families.length; i++)
15.            c.add(families[i]);
16.        add(c);
17.        c = new Choice();
18.        for (int i=0; i<sizes.length; i++)
19.            c.add(""+sizes[i]);
20.        add(c);
21.        setSize(200, 200);
22.    }
23.
24.    public static void main(String[] args)
25.    {
26.        (new FontChoice()).setVisible(true);
27.    }
28. }
```

The resulting GUI is shown in [Figure 15.11](#).



Figure 15.11: Two choices

You can see that the choice component does an excellent job of isolating its parts visually.

Labels

Labels are by far the simplest Java components. They are the only components that cannot be used to gather user input. They just sit there. Often labels appear next to scrollbars, text fields, choices, or other components that do not have their own labels.

A label looks like text on a screen, exactly as if it had been painted there by the `drawString()` method of the `Graphics` class, which you saw in the [previous chapter](#). The following code adds labels to the GUI of the last example in the [previous section](#):

```
1. import java.awt.*;
2.
3. class FontChoiceWithLabels extends Frame
4. {
5.     String[]  families = {"Monospaced", "Serif",
6.                          "SansSerif"};
7.     int[]     sizes    = { 16, 24, 32, 64 };
8.
9.     public FontChoiceWithLabels()
10.    {
11.        setLayout(new FlowLayout());
12.        Label familyLabel = new Label("Font family:");
13.        add(familyLabel);
14.        Choice c = new Choice();
15.        for (int i=0; i<families.length; i++)
16.            c.add(families[i]);
17.        add(c);
18.        Label sizeLabel = new Label("Font size:");
19.        add(sizeLabel);
20.    }
21. }
```



```
20.     c = new Choice();
21.     for (int i=0; i<sizes.length; i++)
22.         c.add(""+sizes[i]);
23.     add(c);
24.     setSize(350, 200);
25. }
26.
27. public static void main(String[] args)
28. {
29.     (new FontChoiceWithLabels()).setVisible(true);
30. }
31. }
```

Lines 12 and 18 construct labels. The constructor takes a single argument, which is the label's text. [Figure 15.12](#) shows this code's GUI.

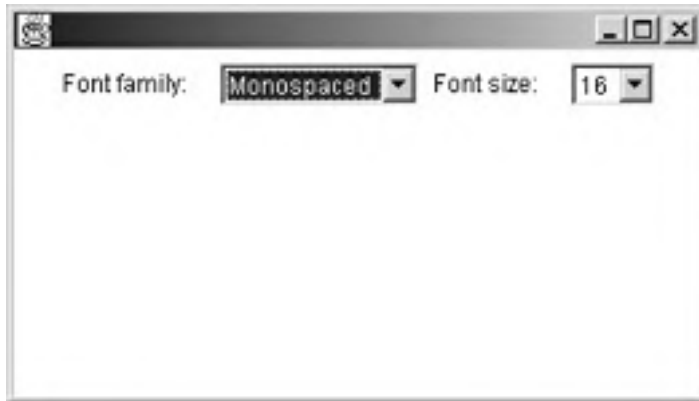


Figure 15.12: Choices with labels

The figure shows a typical use of labels. They rarely appear in isolation. Most often, they are used to give information or instructions about adjacent components.

Menus

Menus have a lot in common with choices:

- They drop down to reveal options.
- The options are arranged vertically.
- They roll back up after a selection is made.

There are also some important differences:

- Menus are attached to a frame's boundary, whereas choices occupy the interior.
- Menus are hierarchical. They can contain submenus, which can contain sub-submenus, and so on. Choices are linear.
- Commercial-grade sites are expected to have menus, which are expected to follow certain conventions. There are no such expectations or conventions for choices.

To insert menus into a frame, follow these steps:

1. Create a menu bar.
2. Create the menus.
3. Attach the menus to the menu bar.
4. Attach the menu bar to the frame.

These steps are all straightforward. The following code creates a frame with a single menu, labeled File. The menu contains three options: Open..., Close, and Exit.

```
1. import java.awt.*;
2.
3. class FrameWithSimpleMenu extends Frame
4. {
5.     String[] options = { "Open...", "Close", "Exit" };
6.
7.     public FrameWithSimpleMenu()
8.     {
9.         // Create the menu bar.
10.        MenuBar mbar = new MenuBar();
11.
12.        // Create the file menu.
13.        Menu fileMenu = new Menu("File");
14.        for (int i=0; i<options.length; i++)
15.            fileMenu.add(options[i]);
```

```
16.
17.     // Populate the menu bar.
18.     mbar.add(fileMenu);
19.
20.     // Attach the menu bar to the frame.
21.     setMenuBar(mbar);
22.
23.     setSize(200, 200);
24. }
25.
26. public static void main(String[] args)
27. {
28.     (new FrameWithSimpleMenu()).setVisible(true);
29. }
30. }
```

Line 10 creates a menu bar. In Java, a frame may have at most one menu bar, to which all menus must be attached. Line 13 creates a File menu. The string passed to the `Menu` constructor appears in the menu bar. Line 15 adds options to the menu. Line 18 attaches the menu to the menu bar. Menus appear on the bar in the order of attachment, from left to right. Finally, line 21 attaches the menu bar to the frame. The result is shown in [Figure 15.13](#).



Figure 15.13: A menu in a menu bar

The `Menu` class has a method called `addSeparator()`, which inserts a horizontal separator bar. You can rewrite the loop at lines 14-15 in the previous example, adding a separator bar between the `Close` and `Exit` items:

```
for (int i=0; i<options.length; i++)
{
    fileMenu.add(options[i]);
    if (i == 1)
        fileMenu.addSeparator();
}
```

The result is shown in [Figure 15.14](#).



Figure 15.14: A menu with a separator

In this example, you've created a menu and added items to it using the `add()` method, passing strings as method arguments. To create a hierarchical menu, add a menu instead of a string:

```
1. import java.awt.*;
2.
3. class FrameWithSubmenu extends Frame
4. {
5.     public FrameWithSubmenu()
6.     {
7.         MenuBar mbar = new MenuBar();
8.
9.         Menu subSubMenu = new Menu("Subsub");
10.        subSubMenu.add("This");
11.        subSubMenu.add("That");
12.        Menu subMenu = new Menu("Sub");
13.        subMenu.add("Here");
14.        subMenu.add("There");
15.        subMenu.add(subSubMenu);
16.        Menu fileMenu = new Menu("File");
17.        fileMenu.add("Open...");
18.        fileMenu.add("Close");
19.        fileMenu.add(subMenu);
20.
21.        mbar.add(fileMenu);
22.        setMenuBar(mbar);
23.
24.        setSize(200, 200);
25.    }
26.
27.    public static void main(String[] args)
28.    {
29.        (new FrameWithSubmenu()).setVisible(true);
30.    }
31. }
```

The GUI appears in [Figure 15.15](#).



Figure 15.15: Hierarchical menus

You can nest menus within menus within menus as much as you want, but don't get carried away. The more complicated your menu structure is, the more difficult it will be for users to find important menu items.

There are many industry-standard conventions that govern the use of menus in GUIs. These are the result of extensive psychological research, as well as many years of practical usage. Here are a few guidelines that are easy to follow:

- There should always be a File menu, and it should occupy the leftmost position in the menu bar. The items New, Open..., and Close should, if present, appear in that order. New should be the first item in the File menu. The Exit item should always be present and should be the last item in the File menu.
- If the application has an Edit menu, it should immediately follow the File menu. The Edit menu should support functions such as Cut, Copy, and Paste.
- If a Help menu is present, it should occupy the rightmost position menu bar.
- Any menu item that causes a new frame or dialog box to be displayed should have three dots following its label. This explains why Open menu items, which typically display file selection dialogs, appear as Open... The three-dots notation is called an *ellipsis*.

These guidelines should be followed in appropriate situations. It isn't necessary to follow them when you're writing code to solve exercises, but keep them in mind whenever you are writing code that is at least moderately complicated and will be used by other people. This book's animated illustrations are all moderately complicated. They are much bigger than exercises and much smaller than commercial applications. They all follow these guidelines.

Text Fields

A text field is a component that displays a single line of text. Unlike labels, text fields respond to keyboard input. To create a text field, use one of the following constructors:

- `TextField(String contents)`
- `TextField(int numColumns)`
- `TextField(String contents, int numColumns)`

The first version creates a text field that is just wide enough to accommodate its contents, which are specified by the string argument. The second version creates a blank text field that is wide enough to accommodate a string of `numColumns` characters. The width is only approximate, since most characters have varying widths when rendered in most fonts. The third version is like the second version, but the text field's contents are initialized to `contents`.

The following application creates two text fields for entering a first and last name. The Last Name text field uses a large non-default font:

```
import java.awt.*;

class TFs extends Frame
{
    public TFs()
    {
        setLayout(new FlowLayout());

        add(new Label("First Name: "));
        TextField tf = new TextField("Livia", 10);
        add(tf);
        add(new Label("Last Name: "));
        tf = new TextField("Soprano", 12);
        Font font = new Font("Monospaced", Font.PLAIN, 24);
        tf.setFont(font);
        add(tf);

        setSize(550, 200);
    }

    public static void main(String[] args)
    {
        (new TFs()).setVisible(true);
    }
}
```

Figure 15.16 shows the GUI.



Figure 15.16: Two text fields

As you can see from the figure, there is something dissonant about having two related text fields with two unrelated fonts. The GUI would be much improved if both fields used the same font, but it illustrates an important point: Text fields can grow to accommodate their fonts. For the moment, we will simply attribute this behavior to the mysterious layout manager, with a promise of a full explanation in the second half of this chapter.

Text Areas

A text area is like a text field, but it can display multiple lines of text. If its contents exceed its height, it can automatically display scrollbars.

The most useful `TextArea` constructor is

```
TextArea(int numRows, int numColumns)
```

The constructor creates a text area with `numRows` rows and `numColumns` columns. Caution: The order of the constructor's arguments might seem backwards. Generally, we are used to specifying first a width and then a height (for example, in the various drawing methods of the `Graphics` class). But `numRows` is a specification of height, and `numColumns` is a specification of width. If you get confused about which comes first, you might try to create a tall, narrow text area and end up with a short, broad one. We say that the dimensions of the text area are specified in *row major* order, which just means that the number of rows comes first.

Here is a very simple program that creates a text area:

```
import java.awt.*;

class TAInnaFrame extends Frame
{
    public TAInnaFrame()
    {
        setLayout(new FlowLayout());
        TextArea ta = new TextArea(10, 30);
        add(ta);
        setSize(550, 220);
    }
}
```

```
}  
  
public static void main(String[] args)  
{  
    (new TAINnaFrame()).setVisible(true);  
}  
}
```

The code is simple, but the text area is not. All we did was create a 10-by-30 text area, as shown in [Figure 15.17](#).



Figure 15.17: A text area

The text area's contents can be changed, either under program control or by user input. A little bit of typing results in [Figure 15.18](#).



Figure 15.18: Multiple checkbox groups

As more text is entered, the contents become taller than the component. When this happens, the text area automatically installs scroll bars, as shown in [Figure 15.19](#).

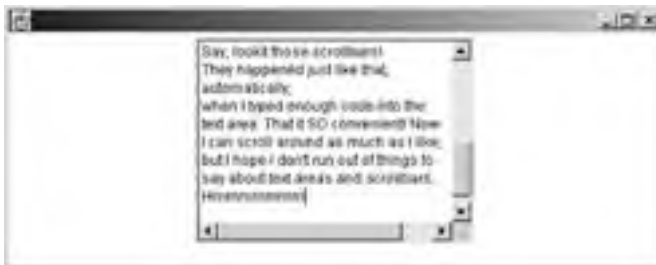


Figure 15.19: A text area with scroll bars

To add text to a text area programmatically, use the `append()` method, which takes a string argument. The string is added to the end of the component's contents. Caution: Text areas do not automatically word-wrap. If you want to add text on a new line, your new text should start with a newline character (`\n`).

Scrollbars

A checkbox has two states, true and false. A choice has several states, one state for each item that might be selected. Both component types are good for situations where users have a limited range of choices. Scrollbars are input devices that come into play when the range of choices is broad. They are most commonly seen in word processors and Web browsers, where they are used to specify the vertical position of a document.

The `java.awt.Scrollbar` class has two main constructors:

- `Scrollbar()`
- `Scrollbar(int orientation)`

The first version creates a vertical scrollbar. The second version creates a scrollbar that is either horizontal or vertical, depending on the value of the `orientation` argument. The class defines two static ints, called `Scrollbar.HORIZONTAL` and `Scrollbar.VERTICAL`. So to construct a horizontal scrollbar, you would call `new Scrollbar(Scrollbar.HORIZONTAL)`.

The following code creates two scrollbars, one in each orientation:

```
import java.awt.*;

class TwoBars extends Frame
{
    public TwoBars()
    {
        setLayout(new FlowLayout());
        add(new Scrollbar()); // Vertical
        add(new Scrollbar(Scrollbar.HORIZONTAL));
        setSize(200, 200);
    }

    public static void main(String[] args)
    {
        (new TwoBars()).setVisible(true);
    }
}
```

There is nothing very exciting about this code. Unfortunately, there is also nothing very exciting about the GUI it creates, as you can see from [Figure 15.20](#).



Figure 15.20: A pair of disappointing scrollbars

Neither of the scrollbars is long enough to be useful. A decent vertical scrollbar should be taller, and a decent horizontal scrollbar should be wider. You can't be effective at setting a component's height or width unless you know about layout managers. Fortunately, we have completed our survey of components, so let's move on.

Layout Managers

The `java.awt` package contains a very useful class called `Container`. Generally, you don't instantiate this class directly. Rather, you instantiate its various subclasses, which include `Frame`. All containers have the following properties:

- They are rectangular.
- They can contain other components, including other containers.
- They use layout managers to determine the locations and positions of the components they contain.

The great thing about layout managers is that you don't have to think about the details of component layout. Each layout manager class imposes a different layout policy on the container it manages. All you have to do is become familiar with the various layout policies available to you. The layout manager will take care of the details.

To change a container's layout policy, you construct an instance of the desired layout manager class. Then you call the container's `setLayout()` method, passing in the layout manager. All the code examples in this chapter have used frames, and the layout manager for a frame is something called a *border layout manager*. The border layout policy is completely inappropriate to what we wanted to do, but a different manager, the *flow layout manager*, is perfect.

This explains why every example constructed an instance of `FlowLayout` and then passed the instance into a `setLayout()` call. Usually this was done in a single line:

```
setLayout(new FlowLayout());
```

To understand layout policies, and therefore to understand why we used flow layout so extensively, you have to understand the concept of *preferred size*. Every component has a preferred size, which a layout manager can either honor or ignore. For components that have text, such as buttons and checkboxes, the preferred size is just large enough to accommodate the component's text. For components without text, the preferred size is arbitrary, which usually is not very good. The preferred size of a scrollbar, for example, is 15x50 pixels.

The Flow Layout Manager

The flow layout manager always honors the preferred size of its container's components. Every component in every figure in this chapter has been its preferred size, because every frame has used a flow layout manager.

When a container uses a flow layout manager, its contained components appear from left to right in the order they were added to the container. There is a gap of five pixels between adjacent components. The cluster of components appears at the top of the container and is centered horizontally. (Horizontal centering is the default. There are other options. See Exercise 6.)

The following code creates three components and uses a flow layout manager to position them in a frame:

```
import java.awt.*;

class SimpleFlow extends Frame
{
    public SimpleFlow()
    {
        setLayout(new FlowLayout());
        add(new Label("ABCDEFGH"));
        add(new Button("Hello"));
        Font f = new Font("SansSerif", Font.BOLD, 24);
        Button btn = new Button("Goodbye");
        btn.setFont(f);
        add(btn);
        setSize(300, 200);
    }

    public static void main(String[] args)
    {
        (new SimpleFlow()).setVisible(true);
    }
}
```

[Figure 15.21](#) shows the code's GUI. Notice how the components are spaced evenly and centered horizontally.



Figure 15.21: Flow layout manager

[Figure 15.22](#) shows the same GUI, after the frame has been made wider. The cluster is still centered.



Figure 15.22: Wider

And [Figure 15.23](#) shows the GUI one last time. Now the frame is too narrow to fit all three components. When this happens, the flow layout manager makes another row. If the frame were even narrower, there would be yet another row.



Figure 15.23: Narrower

You can configure flow layout managers to place their clusters at the left or right of their containers, rather than in the center. You do this by passing an `int` into the `FlowLayout` constructor. If the `int` is `FlowLayout.LEFT`, the cluster will appear at the left; if the `int` is `FlowLayout.RIGHT`, the cluster will appear at the right. [Figure 15.24](#) shows the three-component GUI of the current example, with the `setLayout()` line changed to

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```



Figure 15.24: Left-aligned

The Flow Lab animated illustration lets you experiment with components that are managed by a flow layout manager. To start the application, type `java flow.FlowLab`. [Figure 15.25](#) shows the initial screen.





Figure 15.25: Flow Lab

Flow Lab lets you select a left-, center-, or right-aligning layout manager. You can add buttons, checkboxes, text fields, and labels. Experiment with Flow Lab until you have a good feel for how the layout manager arranges its components. Select left alignment. Add components of a uniform size until the top row is full and a second row is created. How many components are in the top row? Does the number of components in the top row change when you select center or right alignment?

Notice that your selection of layout alignment affects only the main region of the display; the two control panels at the top of the display are not affected. It seems the layout manager you selected is only responsible for part, not all, of the frame. You will see how this is done a little later, in the "Panels" section. But first you have another layout manager to learn about.

The Border Layout Manager

The border layout policy is bizarre at first glance. It only makes sense after you learn what it's good for. So if your reaction to the next few paragraphs is, "This is weird," congratulations. You're right on track.

A border layout manager partially honors and partially ignores the preferred size of its container's components. A preferred size consists of two dimensions: width and height. A border layout manager might ignore one of a component's preferred dimensions while honoring the other. Or both preferred dimensions might be ignored. They are never both honored.

A container that uses a border layout manager may not contain more than five components. The layout manager divides the container into five regions, named North, South, East, West, and Center. Each region may be occupied by zero or one components.

The component in the North region is placed at the top of its container. The component's height is its preferred height; its width is the entire width of the container. [Figure 15.26](#) shows a horizontal scrollbar in the North region of a frame.



Figure 15.26: Scrollbar at North

Here is the code that produced [Figure 15.26](#):

```
1. import java.awt.*;
2.
3. class BarAtNorth extends Frame
4. {
5.     public BarAtNorth()
6.     {
7.         Scrollbar bar=new Scrollbar(Scrollbar.HORIZONTAL);
8.         add(bar, "North");
9.         setSize(200, 100);
10.    }
11.
12.    public static void main(String[] args)
13.    {
14.        (new BarAtNorth()).setVisible(true);
15.    }
16. }
```

The constructor does not call `setLayout()`, because you want to use a border layout manager, which is the default for a frame. In other words, the right kind of layout manager is already there.

Look at line 8. When you add components to a container that uses a border layout manager, you have to pass a second argument to the `add()` method. This is a string that must be North, South, East, West, or Center.

The component at South is attached to the bottom of the container. Otherwise, it is treated like the component at North. Its preferred height is honored, and its width is the entire width of the container.

[Figure 15.27](#) shows a frame with a horizontal scrollbar at North and a text field at South.



Figure 15.27: North and South occupied

The code that produced [Figure 15.27](#) is almost identical to the code that produced [Figure 15.26](#). The difference is that this code

has the following lines before the `setSize()` call:

```
TextField tf = new TextField("Hello");  
add(tf, "South");
```

As you might guess, the components at East and West are attached to the right and left edges of their container, respectively. Their preferred widths are honored. Their heights are the height of the container... almost. They extend all the way up to the top of the container, unless there is a component at North. In that case, they only extend to the bottom of the North component. Similarly, if there is no component at South, the East and West components can extend all the way down to the bottom of the container. But if there is a component at South, the East and West components extend down just to the top of the South component.

There are many combinations of the presence or absence of North or South or East or West components, but [Figure 15.28](#) should make things clear. In the figure, there are components at North, East, and West.



Figure 15.28: North, East, and West occupied

There is no component at South, so the two buttons extend down all the way to the bottom of the container. Since the scrollbar occupies North, the buttons do not extend all the way to the top of the container. They defer to North, extending up to the bottom of the scrollbar. Notice how the buttons have different preferred widths as a result of their different fonts.

So much for North, South, East, and West. The component at Center, if there is one, occupies all the territory that is left over after all other components have been sized and positioned. The white region in [Figure 15.28](#) is the area where there are no components, so the white background of the frame is visible. If the frame had a component at Center, that component would fill the white region exactly. In [Figure 15.29](#), a text area has been added at Center.



Figure 15.29: North, East, West, and Center occupied

Here is the code that produced [Figure 15.29](#):

```
1. import java.awt.*;
2.
3. class NEAndW extends Frame
4. {
5.
6.     public NEAndW()
7.     {
8.         Scrollbar bar=new Scrollbar(Scrollbar.HORIZONTAL);
9.         add(bar, "North");
10.        Button btn = new Button("Me West");
11.        add(btn, "West");
12.        btn = new Button("Me East");
13.        btn.setFont(new Font("Serif", Font.PLAIN, 50));
14.        add(btn, "East");
15.        TextArea ta = new TextArea();
16.        add(ta, "Center");
17.        setSize(600, 400);
18.    }
19.
20.    public static void main(String[] args)
21.    {
22.        (new NEAndW()).setVisible(true);
23.    }
24. }
```

Notice the `TextArea` constructor on line 15. This version is different from the one you were introduced to, where you passed in arguments to specify the number of rows and columns. The no-args version used here is for situations where the text area's size will be determined by the layout manager, so there is no need for you to specify a size.

Panels

Panels are components that divide containers into regions that are smaller and more manageable. The `java.awt.Panel` class extends `java.awt.Container`, so every panel has its own layout manager. You can think of panels as rectangular components that can contain other components, including panels. These in turn can include panels, and so on, so it is possible to create a complex layered hierarchical GUI.

[Figure 15.30](#) shows a frame whose South component is a panel containing three buttons. The panel's only other component is a text area at Center.

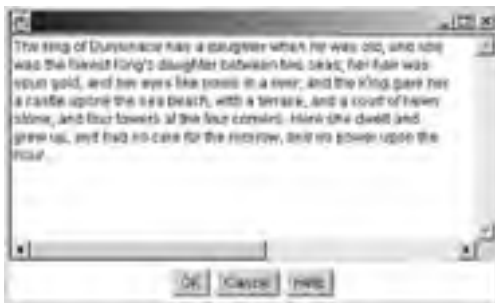


Figure 15.30: A panel in a frame

Here is the code that created [Figure 15.30](#):

```
1. import java.awt.*;
2.
3. class PanelInFrame extends Frame
4. {
5.     public PanelInFrame()
6.     {
7.         Panel pan = new Panel();
8.         pan.add(new Button("OK"));
9.         pan.add(new Button("Cancel"));
10.        pan.add(new Button("Help"));
11.        add(pan, "South");
12.        TextArea ta = new TextArea();
13.        add(ta, "Center");
14.        setSize(400, 250);
15.    }
16.
17.    public static void main(String[] args)
18.    {
19.        (new PanelInFrame()).setVisible(true);
20.    }
21. }
```

When you use panels, you make a lot of calls to the `add()` method of various containers. It's important to keep track of what is being added to what. On lines 8-10, the buttons are added to the panel. On line 11, the panel is added to the frame. On line 13, the text area is added to the frame.

The code has no `setLayout()` calls. The default layout manager for panels is flow. This is confusing, because the default manager for frames is border layout. You have to remember which container type defaults to which layout policy. But in practice, the defaults are usually what you want, so you don't often have to call `setLayout()`.

The flow layout manager makes more sense when you know about panels. Most GUI-based applications consist of a frame that contains a control area and a work area. For example, all Web browsers have a control panel at the top of the display with buttons for going forward, back, home, and so on. Below the control panel is the Web page viewing area. Generally, at the bottom is a status message. When you enlarge the browser, you don't want more space for the controls or the status message; you want a bigger Web page viewing area. The same holds true for most word processors, painting programs, and indeed most programs in general. When the user resizes, it is the main work area below the control area that should do most of the growing.

This is exactly the behavior that you get when you use a frame with a panel at North and some kind of work area at Center. The panel is attached to the top of the frame, and is as wide as the frame. It is as tall as it needs to be to accommodate the components it contains. (That's how the preferred height of a panel is defined.) When the frame becomes wider or narrower, the panel's components are repositioned automatically. When the frame becomes higher or shorter, it is the work area and not the panel that grows or shrinks. At the end of the [next chapter](#), after you have learned how to detect input activity from components, you will work through a final project whose GUI consists of a panel at North and a work area at South.

The Layout Lab animated illustration lets you experiment with hierarchical combinations of containers, layout managers, and components. Layout Lab is designed to let you play with layout ideas without going through the effort of writing code to implement your ideas. To start the program, type `java layout.LayoutLab`. You will see the display shown in [Figure 15.31](#).



Figure 15.31: Layout lab

Initially, the display displays a representation of a frame named Frame0. If you want to change the frame's properties, including its layout manager, click on the Frame0 button. You will see the dialog box shown in [Figure 15.32](#).



Figure 15.32: Layout lab's frame editing dialog

Make sure the frame's layout manager is set to Border. Then dismiss the edit dialog by clicking its Apply button. Now add a component to the frame. Click on the + button. You will see a small dialog that lets you choose a button, a scrollbar, a checkbox, a text field, or a panel. Select Panel, and then click the Apply button. The main window will now look like [Figure 15.33](#).

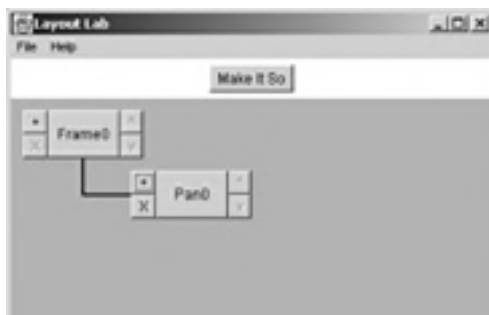




Figure 15.33: Layout Lab with an added panel

Now click on the Pan0 button to edit the properties of the new panel. Since the panel is inside the frame, which uses a border layout manager, one of the panel's properties is its region within the frame (North, South, East, West, or Center). Select South and then click the Apply button.

Now it's time to put a few buttons in the panel. In the main screen, click on the + button. When the little component-chooser dialog appears, select Button and then click Apply. Now the main window will look like [Figure 15.34](#).

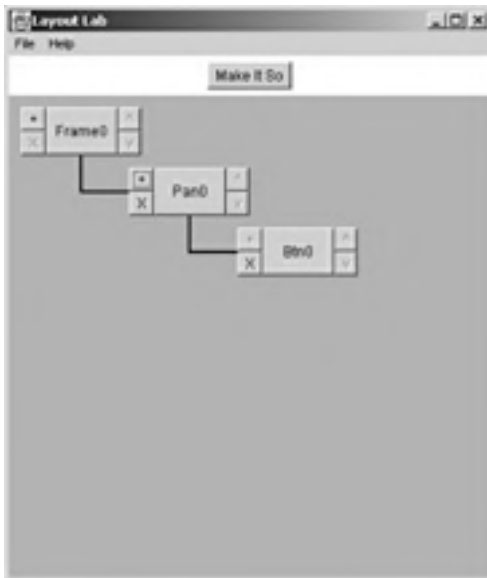


Figure 15.34: A button in a panel in a frame

Edit the button by clicking on Btn0 in the main window. You will see a dialog box that lets you edit the button's location, position, font, and text. Set both X and Y to 500. Set the font to something conspicuously non-default, like SansSerif 36-point bold italic. Set the label to whatever text you like, and click the Apply button.

Now you have created a description of a slightly complicated hierarchical GUI: a button in a panel at the South of a frame. To see what the GUI really looks like, click the Make It So button. You will see a frame that looks like [Figure 15.35](#).





Figure 15.35: Layout Lab makes it so

The button is definitely not 500x500 pixels in size. Is the button's property dialog broken? No. Remember that the panel is using a flow layout manager, which honors the button's preferred size. As you can see in [Figure 15.34](#), the button's preferred size is much smaller than 500x500. In fact, it is just large enough to accommodate the button's text.

Now experiment with layout lab. Try adding more buttons, or other kinds of components, to the panel. Add a panel to the frame's Center. Choose a layout manager for the new panel, add components, including a panel, and add components to *that*. If you want to get rid of a component, click on its X button in the main display. (If the component is a panel, all its contents will be deleted as well. You aren't allowed to delete the frame.) You can click the ^ and v buttons to change the ordering of components in their container.

Play with Layout Lab until you feel comfortable with the idea of components inside a panel that is inside a panel that is inside a frame.

Other Layout Managers

Before we leave the topic of layout managers, it is appropriate to mention that there are other options besides flow and border. The `java.awt` package provides three other managers, called `CardLayout`, `GridLayout`, and `GridBagLayout`. All three are beyond the scope of an introductory book, but you should know that they exist so that you can investigate them if you ever decide you need them.

`CardLayout` allows only one component to be seen at any time. `GridLayout` organizes its container into a grid of rows and columns; each component occupies a single grid location. `GridBagLayout` also creates rows and columns, but it provides many more options than `GridLayout` does.

Several other layout managers (`BoxLayout`, `OverlayLayout`, and `SpringLayout`) are part of the `javax.swing` package. Swing is an alternative to the AWT toolkit. Its components are much more sophisticated than those of AWT.

You can create your own layout manager class. To do this, you implement the `java.awt.LayoutManager` interface. It isn't especially hard once you get the hang of it. The interface only has five methods, and several of them are trivial. Many of this book's animated illustrations display Java source code that is mostly text, with a few scattered text fields or choices that allow you to configure the source code. The data chain lab in [Chapter 13](#) did this. This kind of layout cannot be achieved with any of the standard layout managers, so a new layout manager class was created.

There is one last layout manager option, and it is offered with caution. You can call `setLayout(null)` to operate with no layout manager at all. Then it is your responsibility to set the size and location of every component. You do this by calling the following methods on the components:

`void setLocation(int x, int y)` Sets the component's location (upper-left corner) to (x, y).

`void setSize(int width, int height)` Sets the component's size to width-by-height.

`void setBounds(int x, int y, int width, int height)` Sets the component's location to (x, y) and its size to width-by-height.

The following code uses no layout manager. It creates a 300-by-300 button and positions it at (40, 40):

```
import java.awt.*;

class NullLayout extends Frame
{
    public NullLayout()
    {
        setLayout(null);
        Button btn = new Button("Cancel");
        btn.setSize(300, 300);
        btn.setLocation(40, 40);
        add(btn);
        setSize(400, 400);
    }

    public static void main(String[] args)
    {
        (new NullLayout()).setVisible(true);
    }
}
```

Figure 15.36 shows the GUI.



Figure 15.36: No layout manager

The Layout Lab animated illustration lets you set any container's layout manager to None. Do this to the frame, and add two buttons labeled OK and Cancel. Edit each button's position and size until you like what you see. Get a feel for the ease or difficulty of this task.

The no-layout-manager strategy should be used with caution. As Figure 15.36 shows, it is easy to create a GUI with components of inappropriate size or location. Moreover, when your container has more than a very few components, it is unlikely to look good when the user resizes it.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Suppose you use the following code to create a checkbox:

```
Checkbox cbox = new Checkbox("Ok", true);
```

What is the checkbox's state after you click on it 20,000 times?

2. In the "Checkboxes" section of this chapter, the `Boats` application is 30 lines long. The code isolates literal strings in an array near the top of the listing. You saw how this approach, along with the use of a loop to create the checkboxes, results in more maintainable code. Rewrite the code to eliminate the loop and the string array. In place of the loop in the constructor, just create three checkboxes one by one. How many lines of code does your new application have?
3. This is an extension of Exercise 2. Suppose you need to change the `Boats` application so that instead of offering three sizes (small, medium, and large), it offers ten (rubber duck, sponge, tiny, small, kinda small, medium, kinda large, large, huge, titanic). How does this affect the size of the code as it appears in the "Checkboxes" section of this chapter? How does it affect the size of the code that you wrote for Exercise 2?
4. Write an application that displays a frame with a menu bar. The bar should have the following menus:
 - An Edit menu with items Copy and Cut.
 - A File menu with items Close, Exit, and Open.
 - A Help menu with item Help. Assume that clicking on this item will display a helpful dialog.
 - A Whatever menu with items Stuff and Nonsense. The Nonsense item should be a submenu with items Ordinary Nonsense and Extreme Nonsense.Make sure that your GUI follows the guidelines listed at the end of the "Menus" section.
5. Write a program that creates a GUI that looks like the following illustration. The text in the text area should be set programmatically by a single call to the text area's `append()` method. The call should come directly after the text area is constructed.



6. Using the API page for `java.awt.FlowLayout`, determine how to create a flow layout manager that right-justifies its cluster of components rather than centering it.
7. The `java.awt.Component` class, which is a superclass of `java.awt.Button`, has a method called `setSize(int width, int height)`. The method's documentation says that it resizes the component so that its size is `width` times `height`.

What do you expect the following code to do? First, read the listing and decide on your answer. Then, type in the code and run it. Did you see what you expected to see?

```
import java.awt.*;

class Q7 extends Frame
{
    public Q7()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Abcde");
        btn.setSize(500, 500);
        add(btn);
        setSize(700, 700);
    }

    public static void main(String[] args)
    {
        (new Q7()).setVisible(true);
    }
}
```

8. This entire chapter has been about components that are installed inside containers. The [previous chapter](#) was about painting. What happens if a frame that contains components also has a `paint()` method that paints a part of the screen that is occupied by a component? Write a program that will reveal the answer.

Chapter 16: Events

Overview

Now you know how to create components and lay them out in a GUI. The next step is to learn about *events*, which are the mechanism by which components inform us that they have been used.

A component that does not send events is like a doorbell that does not ring. It looks okay, but it can't make anything happen. It has *look* but no *feel*. In this chapter, you will learn how to make components responsive. This will prepare you for [Chapter 17, "Final Project,"](#) where you will observe in detail a Java application that uses painting, components, events, and all the other programming techniques presented in this book.

Java's original event mechanism was quite limited. It was designed back when it was believed that Java would mostly be used to create applets on Web pages, where space would be limited and GUIs would be simple. It soon became evident, however, that Java was an excellent programming language for domains that had nothing to do with Web pages. As non-Web-based Java applications propagated, GUIs became more complicated, and the current event mechanism was introduced in release 1.1.

The new mechanism is *scalable*, which means it is useful and efficient over a broad range of complexity, from very elementary GUIs to extremely intricate ones. This comes at a price. The event mechanism is not simple. It isn't horribly complicated, but it does consist of several interacting pieces, and it might not make sense until you have seen all the pieces. But hang in there. It will all make sense soon, and when it does, you will have a powerful tool for creating full-fledged GUIs that have both look and feel.

Event-Driven Programs

GUI-based programs are fundamentally different from the applications you saw and wrote prior to [Chapter 14](#) ("Painting"). The earlier programs began execution at the beginning of the `main()` method, ran through the end of `main()`, and that was that. When `main()` was finished, the program was finished. The Java Virtual Machine ceased to exist, and you saw a new prompt in your console window.

Now consider the behavior of an application with a GUI. The following code creates a frame that contains a button and displays a blue circle:

```
import java.awt.*;

public class Xxxx extends Frame
{
    Xxxx ()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Push Me");
        add(btn);
        setSize(300, 300);
    }

    public void paint(Graphics g)
    {
        g.setColor(Color.blue);
        g.fillOval(100, 100, 100, 100);
    }

    public static void main(String[] args)
    {
        (new Xxxx()).setVisible(true);
    }
}
```

[Figure 16.1](#) shows the GUI.

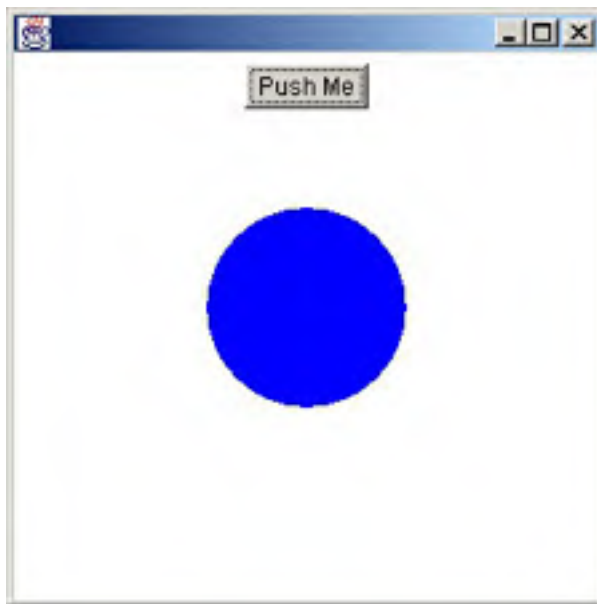


Figure 16.1: A GUI waiting for events

There is something mysterious about this code's behavior, and indeed about the behavior of all the GUI-based applications you saw in [Chapters 14](#) and [15](#). In fact, there are two mysteries, and perhaps you have already wondered about them.

- The `main()` method consists of a single line. After the instance of class `Xxxx` has been constructed and made visible, one would expect the program to terminate. This is not the case. The JVM continues to run (but doing what?) until someone types Ctrl+C into the console window. It is only then that the frame vanishes and the console is ready to accept a new command.
- The application does not call `paint()` anywhere. The method is implemented but never invoked. But something somewhere must have called it, because the circle is right there in the middle of the frame. Who called `paint()`?

Something within the Java Virtual Machine seems to be keeping an eye on things on our behalf, calling `paint()` at the right time and keeping the GUI alive after `main()` finishes. You could almost say the JVM has multiple personalities: One personality to run `main()`, and one to take care of the mysterious GUI behavior.

In fact, the JVM has the computer equivalent of multiple-personality disorder. Don't worry! For computers it's a good thing, because the personalities function together harmoniously. We say that the JVM is *multithreaded*, which means it is capable of

performing more than one task at a time. Each task is called a *thread*. Java's threading capabilities are powerful and intricate. You can write applications that create several or many threads, each performing its own task. Creating your own threads and maintaining harmony among them is far beyond the scope of this book. In order to understand GUI event processing, you don't need to know any details about creating or managing threads. But you do need to know a little bit about what threads are. As you'll see in the [next section](#), my own exposure to threads began many years ago in a barbershop.

Threads

I was eight years old. I was waiting my turn to get my hair cut, and I was reading a Superman comic book. The Man of Steel was entertaining some kids at an orphanage by playing ping-pong against himself. He would serve the ball, and then run at super-speed to the other side of the table to return the ball. Then he would run back to the original side, and so on. Each time he changed sides, he ran so fast that nobody could see him, and he ended up exactly where he had been before. This created the illusion of two identical Supermen.

Computers do something similar. They create the illusion of doing several things simultaneously by rapidly switching from one task to another, many times each second, like Superman running from one side of the ping-pong table to the other. The individual tasks are called *threads*.

Thread support is built into the Java language and the JVM. When you run an application, even a very simple one that just prints out a message from `main()`, there are actually two threads the work. One of these is called the *main thread*. Its job is to execute your `main()` method. Until you began working with GUIs, all behavior of all the applications you wrote came from the main thread.

The second thread is called the *garbage collection thread*. Recall from [Chapter 6, "Arrays."](#) that Java's garbage-collection feature recycles memory from objects and arrays when they can be used no longer. This recycling happens while your program is executing. In other words, the main thread and the garbage collection thread operate simultaneously. You don't have to do anything special to make the garbage collection thread work; it is created automatically by the JVM.

Another thread that is created automatically as part of the JVM infrastructure is the *event dispatch thread*, also known as the *GUI thread*. It is not present in all applications; it appears only in applications with GUIs. The Event dispatch thread knows when the display needs to be redrawn and calls `paint()` at the appropriate moment. As you will see later in this chapter, it is the Event dispatch thread that knows when components have been activated and calls the appropriate methods in the appropriate objects.

The presence of an Event dispatch thread affects the life cycle of the JVM. If an application has no GUI, the JVM terminates when the main thread finishes its work. However, if an Event dispatch thread is present, the JVM continues to run after the main thread is done. The JVM remains in existence until the Event dispatch thread terminates. Typically, this happens when the Event dispatch thread executes a `System.exit()` call.

It is easy to imagine what the JVM is doing while the main thread is alive. Mostly, the JVM is executing the application's bytecode, but now and then the garbage collection thread recycles some memory. But what about a GUI application, where `main()` calls the constructor of a `Frame` subclass, calls `setVisible()` on the constructed object, and then is done? At this point the frame is on the screen, just sitting there. You saw this in numerous examples in the [previous chapter](#). If the frame is doing nothing, and `main()` has terminated, what is the JVM doing?

The answer is: Absolutely nothing! The Event dispatch thread is lurking in the background, waiting for the user to do something that requires attention. For example, if the frame becomes covered by another frame and is subsequently uncovered, the Event dispatch thread will call `paint()` so that the screen can be updated. It is also the job of the Event dispatch thread to notice when user input has occurred, and to respond appropriately by making certain method calls to certain objects.

We say that Java GUI programs are *event-driven*. This means that after some initialization, the programs only act in response to user input. An *event* is a single unit of user input. In the [next section](#), you will learn about Java's simplest type of event.

Action Listeners and Action Events

Java uses many types of events. The simplest is the action event, which is used by buttons and several other components to indicate that simple user input activity has occurred. The other event types are slightly more complicated than action events, but they are used in analogous ways. The nice thing about Java's event mechanism is that once you've learned how to handle one kind of event, it's easy to handle the other kinds.

Every button has a list of objects that are interested in being notified when the button is pushed. These objects are the button's *action listeners*. In general, a *listener* is an object that should be notified when a component is stimulated in some way.

Not all objects are eligible to be a button's action listener. An action listener must implement the `java.awt.event.ActionListener` interface. Note that this interface lives in the `java.awt.event` package, along with all the other classes and interfaces that make up Java's event mechanism. So a GUI application is likely to use the following two import lines:

```
import java.awt.*;
import java.awt.event.*;
```

The first line imports all the component classes; the second imports the event-related classes and interfaces.

The `java.awt.event.ActionListener` interface defines a single method:

```
public void actionPerformed(ActionEvent e);
```

When a button is pressed, the Event dispatch thread constructs an instance of `java.awt.event.ActionEvent`. This is a very simple class that contains a small amount of information about the button activity. Then the Event dispatch thread calls the `actionPerformed()` method of each of the button's action listeners, passing the instance of `ActionEvent` as the method call's argument.

When a button is constructed, its list of action listeners is empty. This explains why none of the buttons created in the example code in the [previous chapter](#) actually caused anything to happen. To add an action listener to a button's list, call the button's `addActionListener()` method, passing as an argument the listener to be added. The listener must implement the `ActionListener` interface.

Here is a class that implements the interface, and so is eligible to be a button's action listener:

```
import java.awt.event.*;

class SimpleActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("The button was pushed.");
    }
}
```

The following code creates a button that uses an instance of `SimpleActionListener` as its action listener:

```
import java.awt.*;

public class UsesListener extends Frame
{
    UsesListener ()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Push Me");
        SimpleActionListener sal = new SimpleActionListener();
        btn.addActionListener(sal);
        add(btn);
        setSize(300, 100);
    }

    public static void main(String[] args)
    {
        (new UsesListener()).setVisible(true);
    }
}
```

[Figure 16.2](#) shows the GUI.



Figure 16.2: A button that sends events

The important thing to notice about [Figure 16.2](#) is that there is nothing important to notice. The button looks perfectly ordinary. There is nothing to tell us that it has feel as well as look. But when you push it, the following message appears in your console:

The button was pushed.

Congratulations! You have now seen your first example of a GUI that has both look and feel.

In addition to the `addActionListener()` method, the `Button` class also has a `removeActionListener()` method, which can be called when a listener no longer wants to get called when the button is pushed.

Note In practice, buttons rarely have multiple action listeners, and `removeActionListener()` is rarely called. In most cases, a button has a single action listener that is added just after the button is constructed and is never removed.

The procedure for writing code with a button that responds to user input can be summarized as follows:

1. Construct a button.
2. Create a listener class that implements the `ActionListener` interface.
3. Construct an instance of your listener class.
4. Call the button's `addActionListener()` method, passing in the instance of your listener class.

The Simple Event Lab animated illustration lets you experiment with buttons and listeners without writing code. Start the program by typing `java events.SimpleEventLab`. You will see the display shown in [Figure 16.3](#).



Figure 16.3: Simple Event Lab: initial screen

The program lets you create simulated buttons and listener classes. You can create simulated instances of the simulated listener classes, click on the buttons, and observe how calls are made to the listeners.

Begin by creating some buttons. Click on Add Button three times. You will see things that look somewhat like buttons, as shown in [Figure 16.4](#).



Figure 16.4: Simple Event Lab with simulated buttons

Now create a (simulated) listener class. In real life, you would do this by writing a class that implements `ActionListener`. In Simple Event Lab, you do it by clicking on Create Listener Class... in the lower part of the frame. The button label ends with dot-dot (officially called *ellipsis*). As you learned in the [previous chapter](#), this means that the button causes a new frame or dialog

box to appear. Indeed, clicking the button brings up a dialog box that lets you choose the name of the class. After you dismiss the dialog, a picture of the class appears at the bottom of the screen, as shown in [Figure 16.5](#).



Figure 16.5: Simple Event Lab with a listener class

The figure shows that a listener class called `GoodListener` has been created. Create your own class, choosing any name you like.

Now it's time to create an instance of the listener class. Click on the picture of the class. You will see a pop-up menu that lets you instantiate the class or delete it. Choose Construct Instance. A simulated instance of the class will appear below the simulated buttons in the main screen, as you can see in [Figure 16.6](#).

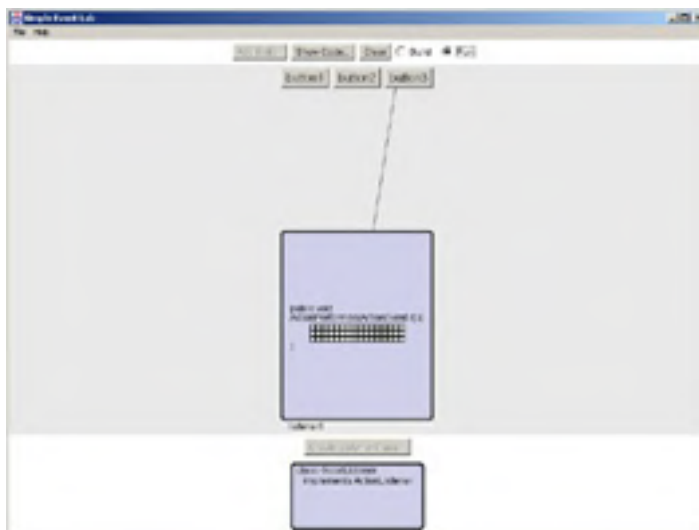


Figure 16.6: Simple Event Lab with a listener object

You can click on the picture of the listener object to change its name or to delete it.

Up to this point, you have simulated the first three steps listed earlier in this section. You have created buttons, you have created a listener class, and you have constructed an instance of the listener class. Now it's time to register the listener object as an action listener of one of the buttons.

Click on one of the simulated buttons. You will see a pop-up menu that invites you to add an action listener or delete the component. Choose Add Action Listener. The cursor will turn into crosshairs. As you move the mouse over the listener object, the object's outline will be highlighted, indicating that you are over a valid listener for the button. Click on the listener. You will see a line connecting the button to the listener.

Now the fun begins. Click the Run button at the top of the screen. The simulated buttons will turn into real buttons, as shown in [Figure 16.7](#).

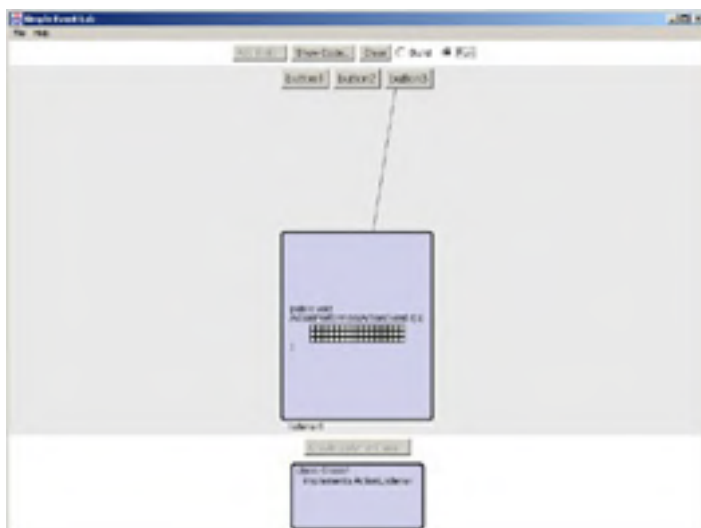


Figure 16.7: Simple Event Lab continued

Now click the button that you connected to the action listener object. The program will show a call being made to the listener's `actionPerformed()` method. The yellow ball represents the `ActionEvent` object.

Click on the Clear button to remove all simulated components, listener classes, and listener objects. Now that you have a clean slate, see if you can repeat the process of connecting a button to a listener without looking at this page.

Experiment with multiple listener classes and multiple listener objects. Can a single listener object be an action listener for more than one button? Can a button have more than one action listener? What does the Show Code... button do?

Getting Information from an Action Event

In the [previous section](#), you were asked to use Simple Event Lab to determine whether a single listener object can be an action listener for more than one button. The answer is yes, as shown in [Figure 16.8](#).

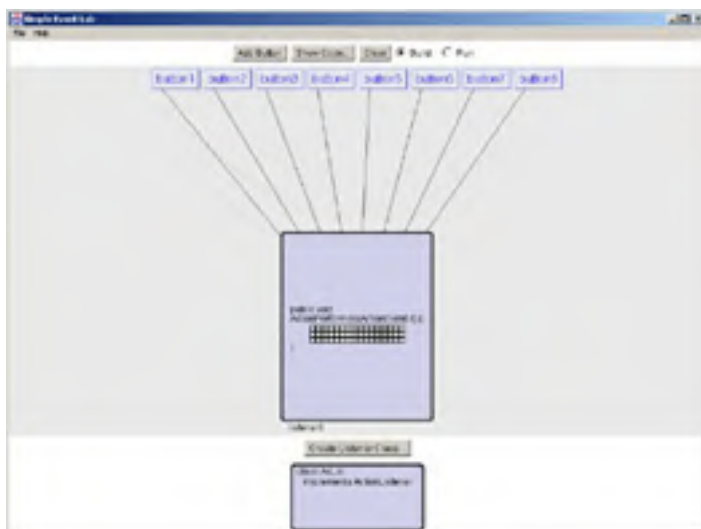


Figure 16.8: One listener object for many buttons

But now there is a problem. Obviously, the code needs to respond differently to different buttons. How does the listener's `actionPerformed()` method know which button was clicked?

The answer is found inside the method's argument. The `ActionEvent` class has a `getSource()` method that returns the button that was clicked. Many `actionPerformed()` methods have a structure that is similar to the following:

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == okButton)
        doOkStuff();
    else if (e.getSource() == cancelButton)
        doCancelStuff();
    else if (e.getSource() == applyButton)
        doApplyStuff();
}
```

The method determines which button was used and responds accordingly. For this to work, the method has to have access to references to the three buttons. The simplest way to make this happen is to put `actionPerformed()` in the frame subclass that creates the buttons. Make sure the frame subclass declares that it implements `ActionListener` (no problem, since it has an

`actionPerformed()` method). Finally, when the buttons are created, the frame subclass itself is registered as their action listener. It looks like this:

```
1. import java.awt.*;
2. import java.awt.event.*;
3.
4. public class ListeningFrame extends Frame
5.     implements ActionListener
6. {
7.     private Button    okButton, cancelButton, applyButton;
8.
9.     ListeningFrame()
10.    {
11.        setLayout(new FlowLayout());
12.        okButton = new Button("Ok");
13.        okButton.addActionListener(this);
14.        add(okButton);
15.        cancelButton = new Button("Cancel");
16.        cancelButton.addActionListener(this);
17.        add(cancelButton);
18.        applyButton = new Button("Apply");
19.        applyButton.addActionListener(this);
20.        add(applyButton);
21.        setSize(300, 100);
22.    }
23.
24.    public void actionPerformed(ActionEvent e)
25.    {
26.        if (e.getSource() == okButton)
27.            doOkStuff();
28.        else if (e.getSource() == cancelButton)
29.            doCancelStuff();
30.        else if (e.getSource() == applyButton)
31.            doApplyStuff();
32.    }
33.
34.    public static void main(String[] args)
35.    {
36.        (new ListeningFrame()).setVisible(true);
37.    }
38. }
```

The `implements ActionListener` statement makes the `ListeningFrame` class eligible to be an action listener for buttons. Lines 13, 16, and 19 register `this` as each button's listener. Recall that `this` is a reference to an object that owns the code being executed. In other words, it's the instance of `ListeningFrame` that is being constructed. The `doOkStuff()`, `doCancelStuff()`, and `doApplyStuff()` methods are omitted.

Here is another example that uses the same design structure. The program plays a version of the game Nim. This game is played by placing 10 coins in a pile. Each player in turn takes one, two, or three coins. The player who takes the last coin is the winner. The GUI consists of four buttons: Take 1, Take 2, Take 3, and Quit. As each player takes a coin, the code prints out the number of remaining coins. [Figure 16.9](#) shows the GUI.

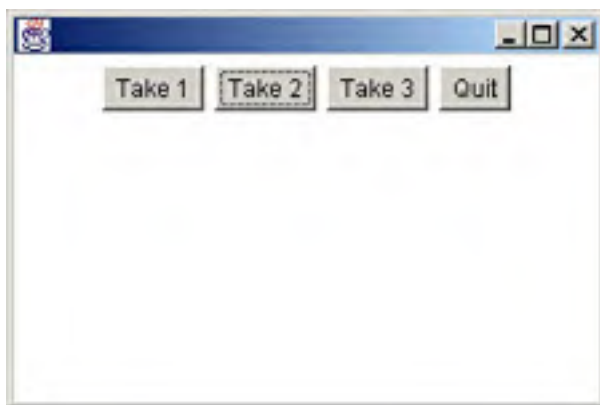


Figure 16.9: Simple Nim GUI

Here's the code:

```
import java.awt.*;
import java.awt.event.*;

public class SimpleNim extends Frame
    implements ActionListener
{
    private Button    btn1, btn2, btn3, quitBtn;
    private int      nCoins;

    SimpleNim()
    {
        nCoins = 10;
        setLayout(new FlowLayout());
    }
}
```

```
        btn1 = new Button("Take 1");
        btn1.addActionListener(this);
        add(btn1);
        btn2 = new Button("Take 2");
        btn2.addActionListener(this);
        add(btn2);
        btn3 = new Button("Take 3");
        btn3.addActionListener(this);
        add(btn3);
        quitBtn = new Button("Quit");
        quitBtn.addActionListener(this);
        add(quitBtn);
        setSize(300, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == quitBtn)
            System.exit(0);

        if (e.getSource() == btn1)
            nCoins -= 1;
        else if (e.getSource() == btn2)
            nCoins -= 2;
        else if (e.getSource() == btn3)
            nCoins -= 3;
        System.out.println(nCoins + " left.");
    }

    public static void main(String[] args)
    {
        (new SimpleNim()).setVisible(true);
    }
}
```

The `actionPerformed()` method first determines if the event source was the Quit button. If so, `System.exit()` is called to terminate the program. In event-driven programming, calling `System.exit()` in response to user input is the appropriate way to end a program. If the event came from one of the Take buttons, the coin count `nCoins` is decremented and the remaining value is printed out.

This application works as an example of how to process events, but it is certainly no improvement over a pile of coins. (Unless you don't have 10 coins. But if you don't have 10 coins, you probably can't afford a computer.) The situation points out an important principle of GUI design, which is violated all too often on the World Wide Web: Only create a GUI if it makes life better.

In the [next section](#), you will see the last example improved on in several ways. You might not think the final version is better than a pile of coins, but you will certainly find it an improvement over the original version. And, more importantly, you will learn some important techniques for creating useful GUIs.

Improving the GUI

In this section, the `SimpleNim` application will be improved in three stages. To keep life simple, the Nim Lab program on your CD-ROM gives you easy access to all four versions (the original and the three improvements). To run Nim Lab, type `java events.NimLab`. You will see the display shown in [Figure 16.10](#).

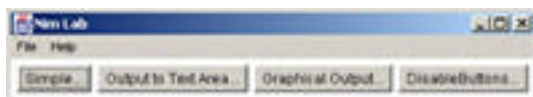
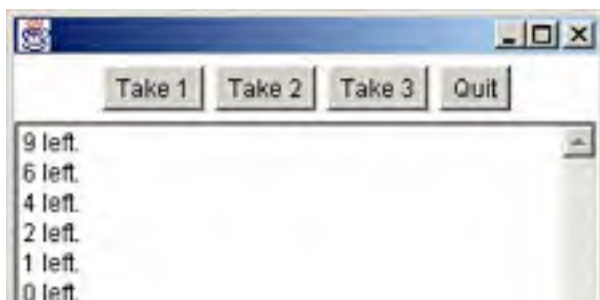


Figure 16.10: Nim Lab

Each improvement will illustrate a general principle of GUI design. The first principle is that the results of user input activity should appear near where the activity happened. In this way, cause and effect are visually related. (The cause is the input, and the effect is the resulting change to the screen.) In the `SimpleNim` version, you clicked buttons in the GUI, but your output appeared at the console from which you ran the program. This is inconvenient, because you have to keep moving your eyes back and forth.

It would be better if the output could happen in the GUI. For this, you will use a text area. The `TextArea` class has a method called `append()` that appends text the component's contents, so let's modify the `actionPerformed()` method so that it calls `append()` rather than `System.out.println()`.

[Figure 16.11](#) shows the GUI after a game has been played.



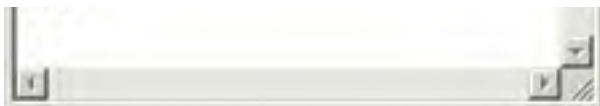


Figure 16.11: Nim, with output to a text area

Here is the code:

```
import java.awt.*;
import java.awt.event.*;

public class TextAreaNim extends Frame
    implements ActionListener
{
    private Button    btn1, btn2, btn3, quitBtn;
    private TextArea  ta;
    private int       nCoins;

    TextAreaNim()
    {
        nCoins = 10;

        Panel controls = new Panel();
        btn1 = new Button("Take 1");
        btn1.addActionListener(this);
        controls.add(btn1);
        btn2 = new Button("Take 2");
        btn2.addActionListener(this);
        controls.add(btn2);
        btn3 = new Button("Take 3");
        btn3.addActionListener(this);
        controls.add(btn3);
        quitBtn = new Button("Quit");
        quitBtn.addActionListener(this);
        controls.add(quitBtn);
        add(controls, "North");

        ta = new TextArea(40, 20);
        add(ta, "Center");
        setSize(300, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == quitBtn)
            System.exit(0);

        if (e.getSource() == btn1)
            nCoins -= 1;
        else if (e.getSource() == btn2)
            nCoins -= 2;
        else if (e.getSource() == btn3)
            nCoins -= 3;
        ta.append(nCoins + " left.\n");
    }

    public static void main(String[] args)
    {
        (new TextAreaNim()).setVisible(true);
    }
}
```

The frame uses a border layout manager. There is a panel (`controls`) at North containing the buttons. The text area is at Center. Thus, when you make the frame bigger (try it!), most of the new space goes to the text area.

The original line

```
System.out.println(nCoins + " left.");
```

has been replaced by

```
ta.append(nCoins + " left.\n");
```

Notice the newline character (`\n`) in the new version. When you call `System.out.println()`, a newline is printed automatically. This does not happen when you call `append()` on a text area, so you have to provide your own newline.

This version is definitely an improvement. You no longer have to look up to do input and look down to read output. But the output is pure text.

The next principle of GUI design that we will apply is this: Show me, don't tell me. Our next improvement will be to draw coins on the screen, rather than displaying text that merely tells you about coins. This is not a book on graphic design, so the coins will just be filled circles. But the code will show what you could do if you were working with a graphics designer who provided you with code for painting exquisitely detailed coins.

Figure 16.12 shows the initial state of the new version.

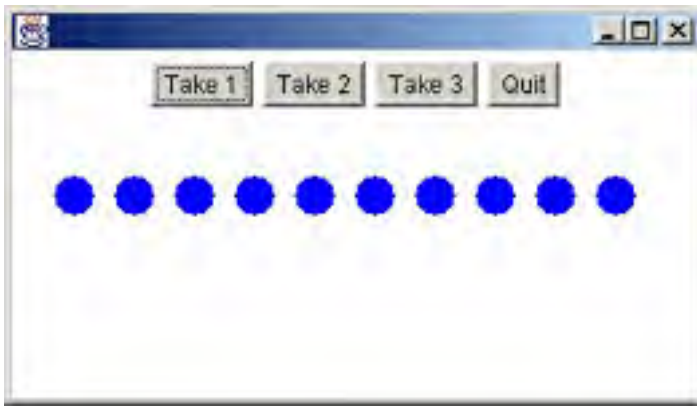


Figure 16.12: Nim with graphical output

Figure 16.13 shows the GUI after a few coins have been taken.

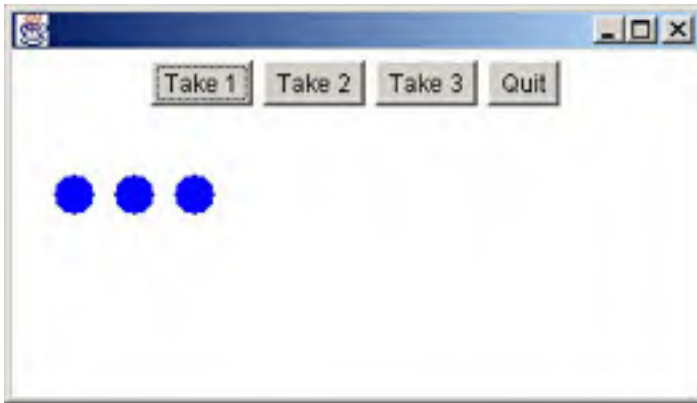


Figure 16.13: Nim with graphical output, game in progress

Figures 16.12 and 16.13 dramatically show that pictures are better than words. Here's the code:

```
import java.awt.*;
import java.awt.event.*;

public class GraphicOutputNim extends Frame
    implements ActionListener
{
    private Button    btn1, btn2, btn3, quitBtn;
    private int      nCoins;

    GraphicOutputNim()
    {
        nCoins = 10;

        Panel controls = new Panel();
        btn1 = new Button("Take 1");
        btn1.addActionListener(this);
        controls.add(btn1);
        btn2 = new Button("Take 2");
        btn2.addActionListener(this);
        controls.add(btn2);
        btn3 = new Button("Take 3");
        btn3.addActionListener(this);
        controls.add(btn3);
        quitBtn = new Button("Quit");
        quitBtn.addActionListener(this);
        controls.add(quitBtn);
        add(controls, "North");
        setSize(350, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == quitBtn)
            System.exit(0);

        if (e.getSource() == btn1)
            nCoins --;
        else if (e.getSource() == btn2)
            nCoins -= 2;
        else if (e.getSource() == btn3)
            nCoins -= 3;
        repaint();
    }
}
```

```
    }  
  
    public void paint(Graphics g)  
    {  
        int x = 25;  
        int y = 85;  
  
        g.setColor(Color.blue);  
        for (int i=0; i<nCoins; i++)  
        {  
            g.fillOval(x, y, 20, 20);  
            x += 30;  
        }  
    }  
  
    public static void main(String[] args)  
    {  
        (new GraphicOutputNim ()).setVisible(true);  
    }  
}
```

The text area is gone. The line at the end of `actionPerformed()`, which originally called `System.out.println()` and then called the text area's `append()` method, is now the following:

```
repaint();
```

The `repaint()` method causes two things to happen:

1. The frame's interior is cleared to its background color. (Components contained in the frame are not affected.)
2. A call is made to the frame's `paint()` method.

Whenever you want a display to be refreshed in response to GUI input, calling `repaint()` is the best approach. It would be beyond the scope of this book to explain why. Here is an oversimplified explanation: When you call `repaint()`, eventually your `paint()` method will be called, at an appropriate time, with an appropriate `Graphics` argument. You never need to call `paint()` directly. You are always better off calling `repaint()` and letting the environment call `paint()`.

The `paint()` method uses a loop to draw the appropriate number of blue-filled circles, based on the value of `nCoins`. The variable `x` determines the horizontal position of each circle's bounding box. It is increased by 30 each time a circle is drawn. To see the program in action, run Nim Lab and select Graphical Output...

Now let's make one last improvement to enforce what is perhaps the most important GUI principle of all. If you pay attention to the other principles, you might create a great GUI. But if you ignore the most important principle, you will certainly create a poor GUI.

Here's the most important principle: A GUI should *never* let a user perform illegal input.

The latest Nim version violates this rule. To see this, run Nim Lab and select Graphical Output.... Click the Take 3 button three times. Now there is only one coin left, but the GUI will let you take two or three coins. This should not be allowed. You also should not be allowed to take three coins if there are two coins left.

There are two ways to make illegal input impossible. At the appropriate time, the buttons can be either removed or disabled. Removing the buttons may sound like a good idea (after all, you can't push a button that isn't there), but extensive research has shown that users are uncomfortable with GUIs whose components pop in and out of existence. This approach creates too much movement in the peripheral field of vision. The commonly accepted technique is to disable components that should not be used. The components are still visible, but they are unresponsive. A disabled component has a slightly different appearance. It is somewhat grayer than its enabled counterpart. [Figure 16.14](#) shows two buttons. The first is enabled, the second is disabled.



Figure 16.14: Enabled and disabled buttons

In the previous version of the Nim GUI, the Take buttons are enabled only if there are enough coins left. [Figure 16.15](#) shows the program when one coin remains.



Figure 16.15: Nim with disabled buttons

Notice that the Take 2 and Take 3 buttons are disabled. To enable or disable any component, call its `setEnabled()` method. The method takes a boolean argument: `true` to enable, `false` to disable. Here's the code:

```
import java.awt.*;
import java.awt.event.*;

public class DisablingNim extends Frame
    implements ActionListener
{
    private Button    btn1, btn2, btn3, quitBtn;
    private int      nCoins;

    DisablingNim()
    {
        nCoins = 10;

        Panel controls = new Panel();
        btn1 = new Button("Take 1");
        btn1.addActionListener(this);
        controls.add(btn1);
        btn2 = new Button("Take 2");
        btn2.addActionListener(this);
        controls.add(btn2);
        btn3 = new Button("Take 3");
        btn3.addActionListener(this);
        controls.add(btn3);
        quitBtn = new Button("Quit");
        quitBtn.addActionListener(this);
        controls.add(quitBtn);
        add(controls, "North");
        setSize(350, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == quitBtn)
            System.exit(0);

        if (e.getSource() == btn1)
            nCoins -= 1;
        else if (e.getSource() == btn2)
            nCoins -= 2;
        else if (e.getSource() == btn3)
            nCoins -= 3;

        if (nCoins < 3)
            btn3.setEnabled(false);
        if (nCoins < 2)
            btn2.setEnabled(false);
        if (nCoins < 1)
            btn1.setEnabled(false);

        repaint();
    }

    public void paint(Graphics g)
    {
        int x = 25;
        int y = 85;

        g.setColor(Color.blue);
        for (int i=0; i<nCoins; i++)
        {
            g.fillOval(x, y, 20, 20);
            x += 30;
        }
    }

    public static void main(String[] args)
    {
        (new DisablingNim()).setVisible(true);
    }
}
```

The new code appears at the end of `actionPerformed()`:

```
if (nCoins < 3)
    btn3.setEnabled(false);
if (nCoins < 2)
    btn2.setEnabled(false);
if (nCoins < 1)
    btn1.setEnabled(false);
```

Further improvements to the GUI are possible. (See [Exercise 5](#) at the end of this chapter.)

But enough about Nim. At this point, you know how to respond to GUI input from buttons. It will be easy to move on to responding to other component types.

Team LIB

← PREVIOUS

NEXT →

Events from other Components

In [Chapter 15](#), you learned how to create a variety of component types:

- Buttons
- Check boxes
- Choices
- Labels
- Menus and menu items
- Scrollbars
- Text areas
- Text fields

Now you will learn how to respond to user input activity on each type of component. You already know how to respond to buttons. Labels do not send events. In the rest of this chapter, you will learn how to detect events from the other component types.

Check Boxes, Choices, and Item Events

In this section, you'll learn how to respond to activity from check boxes and choices. As a reminder, [Figure 16.16](#) shows a check box and a choice.

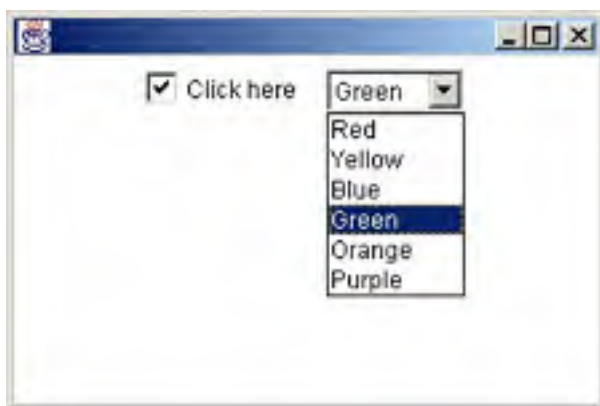


Figure 16.16: Check box and choice

Check boxes and choices don't have action listeners, but they have something similar: item listeners. An object that wants to be notified when activity happens in a check box or a choice must implement the `java.awt.event.ItemListener` interface. This interface defines one method:

```
public void itemStateChanged(ItemEvent e);
```

When a check box or choice is activated, an `itemStateChanged()` call is made to each of its item listeners. You can call `addItemListener(ItemListener x)` to add an item listener to a check box's or choice's list. You can call `removeItemListener(ItemListener x)` to remove an item listener from a check box's or choice's list.

Within an `itemStateChanged()` method, you can determine which component was activated by calling the `ItemEvent`'s `getSource()` method, just as you would call `getSource()` on an `ActionEvent` in an `actionPerformed()` method. (In fact, `ActionEvent`, `ItemEvent`, and the other event classes you will learn about in this chapter all inherit `getSource()` from a superclass that they all extend.)

The following code builds a GUI that contains a check box, a choice, and a text field. When the check box or the choice are activated, the text field displays an appropriate message.

```
import java.awt.*;
import java.awt.event.*;

public class CboxAndChoice extends Frame
    implements ItemListener
{
    private Checkbox    cbox;
    private Choice      ch;
    private TextField   tf;

    CboxAndChoice()
    {
        setLayout(new FlowLayout());
        cbox = new Checkbox("Click here");
        cbox.addItemListener(this);
        add(cbox);
```



```
ch = new Choice();
ch.add("Red");
ch.add("Yellow");
ch.add("Blue");
ch.add("Plaid");
ch.add("Paisley");
ch.addItemListener(this);
add(ch);

tf = new TextField(25);
add(tf);

setSize(475, 75);
}

public void itemStateChanged(ItemEvent e)
{
    if (e.getSource() == cbox)
        tf.setText("Checkbox: " + cbox.getState());
    else
        tf.setText("Choice: " + ch.getSelectedIndex());
}

public static void main(String[] args)
{
    (new CboxAndChoice()).setVisible(true);
}
}
```

The `itemStateChanged()` method calls the event's `getSource()` method to determine which component was activated. The `getState()` method of `Checkbox` returns `true` if the component is checked, and `false` if it is not checked. The `getSelectedIndex()` method of `Choice` returns the position (counting from 0) of the component's selected item.

[Figure 16.17](#) shows the GUI.



Figure 16.17: Receiving events from a check box and a choice

By now, you probably get the feel of it. Components have lists of listeners. When the components are activated, method calls are made to the listeners.

That's about it. You'll probably have an easy time with the next several sections.

Text Fields and Text Areas

Text fields and text areas both send text events to text listeners. The events are sent each time a user types a keystroke. The `TextListener` interface defines one method:

```
public void textValueChanged(TextEvent e);
```

To add an object to a text field's or text area's list of text listeners, call the component's `addTextListener()` method, passing in the listener object.

Text fields (but not text areas) can also send action events to action listeners. This happens when the user presses the Enter key.

We won't work through a detailed code example, because if you understand how to handle action and item events, handling text events should be obvious. Instead, let's step back for a moment and look at the big picture.

A Java GUI consists of a number of components of various types. Each component may have zero, one, or multiple listeners for each event type that the component supports. When a component is activated, the Event dispatch thread calls the appropriate method of each listener.

The Event Lab animated illustration lets you experiment with multiple component, listener, and event types, without writing any code. Event Lab is an extension of Simple Event Lab. In addition to buttons, you can create check boxes, choices, and text fields. When you create a listener class, you select which listener interfaces it will implement. (Your choices are `ActionListener`, `ItemListener`, and `TextListener`. Remember that classes are allowed to implement more than one interface, so listener classes are allowed to implement more than one listener interface.)

Start the program by typing `java events.EventLab`. You control the program just as you did Simple Event Lab. [Figure 16.18](#) shows Event Lab with a fairly complicated configuration.

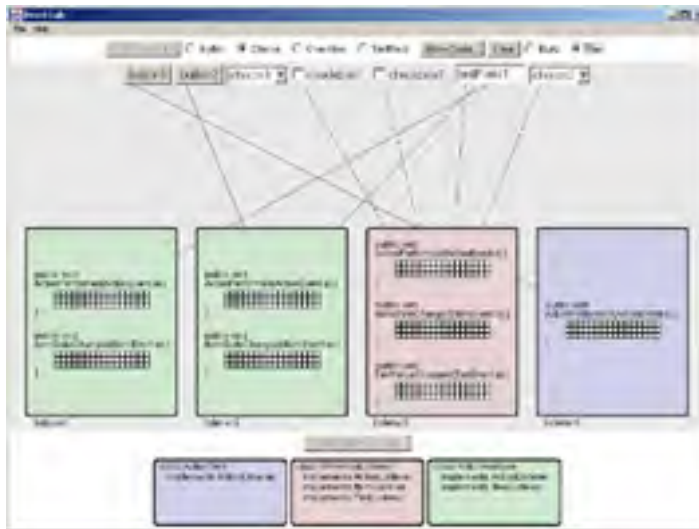


Figure 16.18: Event Lab

Configure Event Lab with your own complicated setup. Then click the Run button. The simulated components will become real. Activate your buttons, check boxes, choices, and text fields until you have a good feel for how various component types send various event types to various listeners in an event-driven GUI program.

Events from Menus

In [Chapter 15](#), you saw the simplest way to populate menus. It looked something like this:

```
Menu fileMenu = new Menu("File");
fileMenu.add("Open...");
fileMenu.add("Close");
...
```

The `add()` calls created individual menu items. That approach was good for showing you what Java menus look like, but there is a better way if you want to receive event notification from the menu items. The preceding code can be rewritten as follows:

```
Menu fileMenu = new Menu("File");
MenuItem openMI = new MenuItem("Open...");
fileMenu.add(openMI);
MenuItem closeMI = new MenuItem("Close");
fileMenu.add(closeMI);
...
```

Like buttons, menu items send action events to action listeners. So to add `menuListener` to `openMI`'s list of action listeners, you would call

```
openMI.addActionListener(menuListener);
```

There's no need to present a detailed example, because the code is so similar to the code you've already seen that handles action events from buttons.

Scrollbars and Adjustment Events

Scrollbars send adjustment events to adjustment listeners. Adjustment listeners implement the `java.awt.event.AdjustmentListener` interface. Once again, we have a listener interface that defines a single method:

```
public void adjustmentValueChanged(AdjustmentEvent e);
```

An object gets added to a scrollbar's listener list via a call to the `addAdjustmentListener()` method. The following code receives adjustment notification from a scrollbar, and reports the scrollbar's value to a text field. The code uses the `getValue()` method of the `Scrollbar` class. The return type is int:

```
import java.awt.*;
import java.awt.event.*;

public class BarAndTF extends Frame
    implements AdjustmentListener
{
    private Scrollbar    bar;
    private TextField    tf;

    BarAndTF ()
    {
        bar = new Scrollbar(Scrollbar.HORIZONTAL);
        bar.addAdjustmentListener(this);
        add(bar, "North");
        Panel pan = new Panel();
        tf = new TextField("          ");
        pan.add(tf);
        add(pan, "South");

        setSize(300, 100);
    }
}
```

```
public void adjustmentValueChanged(AdjustmentEvent e)
{
    tf.setText("Value = " + bar.getValue());
}

public static void main(String[] args)
{
    (new BarAndTF()).setVisible(true);
}
}
```

The GUI is shown in [Figure 16.19](#).



Figure 16.19: Scrollbar and text field

Now you know how to respond to events from all the component types you learned about in [Chapter 15](#). In the [next chapter](#), which finishes this book, you will work through a detailed final project that draws from everything you have learned so far, from [Chapter 1](#) through the period at the end of this sentence.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Write a program that displays a frame. The frame's `paint()` method should draw something simple. The application should also maintain a count of the number of times `paint()` is called. This count should be printed out every time `paint()` is called. Execute your application, and use it to help determine whether `paint()` is called when:

The application starts up.

The frame is minimized/iconified.

The frame is restored to normal size after being minimized/iconified.

The frame is restored to normal size after being minimized/iconified.

The frame is moved.

The frame is partially covered by another frame.

The frame is uncovered.

2. Every Java thread is represented by an instance of the `java.lang.Thread` class. You can get a reference to the currently running thread by calling the `currentThread()` static method of the `Thread` class. Threads have names. The class has a method called `getName()`, which returns the name as a string. So you can print out the name of the current thread by calling

```
System.out.println(Thread.currentThread().getName());
```

Write a simple frame application that makes this call in its `main()` method and in its `paint()` method. Verify that `main()` and `paint()` are executed in different threads.

3. Write an application that adds the same action listener to a button *twice*. For example, if `myButton` is the button and `myListener` is the action listener, your code would contain the following lines:

```
myButton.addActionListener(myListener);  
myButton.addActionListener(myListener);
```

Your listener's `actionPerformed()` method should print out a message to tell you that it got called. If you press the button once, do you expect the message to be printed out once or twice? Run your application to see if you guessed right.

Of course, in real life there would never be a good reason for doing this. But you might do it by accident. For example, you might paste the line into your source code twice by accident. So it's good to know in advance what the symptom will be, so that you can recognize it and fix the problem if it ever comes up.

4. Suppose a class has an `actionPerformed()` method, as specified by the `ActionListener` interface, but the class does not state that it implements the interface. Can an instance of the class be used as a button's action listener?
5. Run Nim Lab by typing `java events.NimLab`. Select Disable Buttons... and play the game. This version is the result of three rounds of improvements made to the original program. What additional improvements can you suggest? Think about how the game could be modified to make the GUI easier and more natural.
6. The various event classes (`ActionEvent`, `ItemEvent`, etc.) all inherit the `getSource()` method from a superclass. Use the API pages to determine the name of that superclass.
7. Write an application with a GUI that contains a choice and a text area. When the choice is activated, a message should be written to the text area, stating the choice's selected index.

Suggested design: Your frame should contain a panel (at North) that contains the choice. The text area should be at South. If you need a guideline, the `TextAreaNim` program in the "Improving the GUI" section has a similar structure.

8. Write an application with a GUI that contains a text field and a text area. When the user presses the Enter key in the text field, the text field's contents should be copied into text area, followed by a newline character.

Your event-handling code will need to retrieve the contents of the text field. You do that by calling the text field's `getText()` method, which returns a string.

Suggested design: Your frame should contain a panel at North that contains the text field. The text area should go at Center.

Chapter 17: Final Project

You made it! With the presentation on event handling in the [previous chapter](#), you have finished your from-the-ground-up introduction to the Java programming language. You now know a *lot* about Java, and in this chapter you'll prove it. You will observe the development of a substantial programming project, and it will all make sense. The project will draw on the information you learned in every other chapter of this book. It includes a GUI that paints, uses components, and sends out events. Classes will be extended and interfaces will be implemented. Exceptions will be thrown and caught.

This chapter doesn't just walk you through a finished, polished program. That would be like dissecting an animal in high-school biology class. Seeing something grow and develop is better than studying something that's dead. So for each piece of the project, you will see not just the final product, but also the living process that culminates in the finished, polished program.

Description of the Project

We will create a GUI that displays Java source code in an easy-to-read format. The user will be able to choose any .java file. Most of the code will appear in black letters, but line comments and Java keywords will appear in different colors, to be specified by the user.

[Figure 17.1](#) shows the application in action. It is displaying one of the source files of the project.



Figure 17.1: Final Project

The Show lines check box draws horizontal lines to make the text more readable. [Figure 17.2](#) is the same code, with lines.

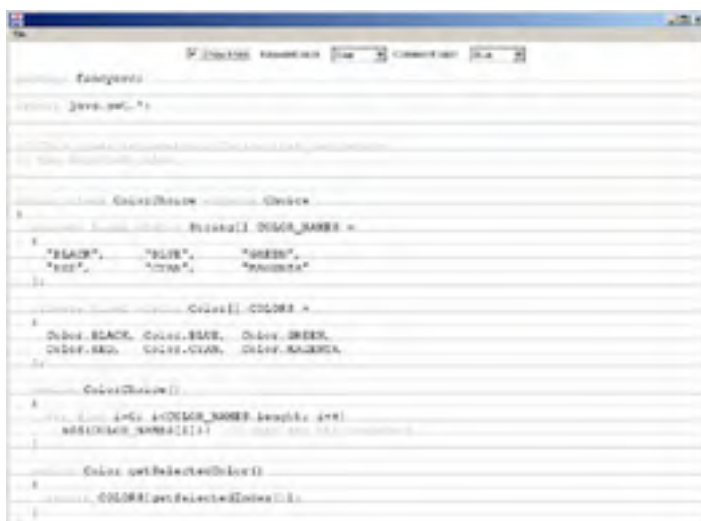


Figure 17.2: Final Project, with lines

The black-and-white figures don't communicate the dramatic effect of multicolored source code.

Now is a good time to run the actual project code, so you can get familiar with what you'll be doing in the rest of this chapter. Type `java fancysrc.FancySrcFrame`. You'll see the GUI controls shown in the figures, above a blank display area. To display

some code, select Open... in the File menu. You will see a file chooser. Use the chooser to select one of the source files on the CD-ROM, or one of your own Java source files.

Now let's get to work.

Team LIB

PREVIOUS **NEXT**

Building the Pieces

The overall structure of the project code will be familiar to you. We will create a subclass of `java.awt.Frame`, called `FancySrcFrame`, in package `fancysrc`. The frame will have a panel at North, containing various components. The `FancySrcFrame` class will be the event listener for all events from all components.

We will divide the work into five pieces. We will develop each piece in turn before assembling everything into the final product. The five pieces are:

- The menu
- The file-specification code
- The color-specification code
- The display area
- The painting code

Each piece of the project is discussed in its own section.

The File Menu

Over the years, the software industry has created a substantial number of conventions for interacting with GUI-based programs. Every GUI is different, but they can all be approached in the same way, with the same reasonable expectations. This is enormously beneficial to the community of software users (that's us), because it reduces the amount of time we have to spend learning to use a new program.

The automobile industry is in a similar situation. If you know how to drive a car, you pretty much know how to drive *every* car. If this were not the case, car rental would be even more stressful than it already is.

One of the standard GUI practices is to install a menu bar in every program's main frame. The leftmost menu is a File menu, whose last item is Exit. In our case, the File menu will have an Open... item. Here we won't worry about how to actually open a file. That's covered in the [next section](#), "Specifying a File." For the moment, our concern is to construct a menu bar with a File menu.

Building and responding to a menu requires techniques that were presented in [Chapters 15, "Components,"](#) and 16, ["Events."](#) The constructor for our main application class (`FancySrcFrame`, in package `fancysrc`) will build the menu.

When you write code that builds menus, you might find it helpful to draw a diagram like the one in [Figure 17.3](#).



Figure 17.3: Menu schematic

A menu schematic might be trivial for the project at hand, but it makes life much easier if you are creating complicated menu bars, with many menus and submenus. After you write your menu code, you can test all the menus to make sure they match your schematic.

[Chapter 15](#) presented a list of steps for building a menu structure:

1. Create a menu bar.
2. Create the menus.
3. Attach the menus to the menu bar.
4. Attach the menu bar to the frame.

Here is some code that builds the menu structure:

```
1. MenuBar mbar = new MenuBar();
2. Menu fileMenu = new Menu("File");
3. openMI = new MenuItem("Open...");
4. openMI.addActionListener(this);
5. fileMenu.add(openMI);
6. exitMI = new MenuItem("Exit");
7. exitMI.addActionListener(this);
8. fileMenu.add(exitMI);
9. mbar.add(fileMenu);
10. setMenuBar(mbar);
```

Line 1 creates a menu bar. Lines 2-8 create a menu. Line 9 attaches the menu to the menu bar, and line 10 attaches the menu bar to the frame. Assume the current class implements the `java.awt.event.ActionListener` interface, so `this` is a legal argument to the `addActionListener()` calls in lines 4 and 7.

The variables `mbar` and `fileMenu` are declared within the constructor. (Remember, all the preceding code goes in the

FancySrcFrame constructor.) However, `openMI` and `exitMI` will be declared as variables of the `FancySrcFrame` class. You'll see why shortly. Meanwhile, can you guess? (Hint: It has something to do with event handling.)

Now that we have a small chunk of code, let's test it. We'll embed it in a test class called `MenuTest`. This class implements `ActionListener`, so that lines 4 and 7 will compile. If the code works, we can later copy it verbatim from the test program into our final project code.

Here is the source for `MenuTest`:

```
package fancysrc;

import java.awt.*;
import java.awt.event.*;

class MenuTest extends Frame implements ActionListener
{
    private MenuItem openMI, exitMI;

    MenuTest()
    {
        MenuBar mbar = new MenuBar();
        Menu fileMenu = new Menu("File");
        openMI = new MenuItem("Open...");
        openMI.addActionListener(this);
        fileMenu.add(openMI);
        exitMI = new MenuItem("Exit");
        exitMI.addActionListener(this);
        fileMenu.add(exitMI);
        mbar.add(fileMenu);
        setMenuBar(mbar);
        setSize(250, 100);
    }

    public void actionPerformed(ActionEvent e)
    {
    }

    public static void main(String[] args)
    {
        (new MenuTest()).setVisible(true);
    }
}
```

The `actionPerformed()` method doesn't do anything, because for now we just want to check the structure of the menu. We are testing look, not feel. If you want to run `MenuTest`, it's on your CD-ROM. Just type `java fancysrc.MenuTest`. It looks like [Figure 17.4](#).



Figure 17.4: Teting the menu's look

The figure matches the menu schematic from [Figure 17.3](#), so apparently the code is good.

Now let's add code to respond to menu activation, so we can test feel as well as look. We just need to put some `println()` calls in the `actionPerformed()` method. Later we'll replace the calls with code that actually opens a file or exits the program.

Here's the new version of `actionPerformed()`:

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == openMI)
        System.out.println("OPEN Menu item");
    else
        // Must be the "Exit" menu item.
        System.out.println("EXIT Menu item");
}
```

When the test program is run and the two menu items are activated one after another, the output is

```
OPEN Menu item
EXIT Menu item
```

The output shows that the menu item action events are being handled correctly.

Implementing the exiting code is trivial. We just insert a call to `System.exit()`:


```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == openMI)
        System.out.println("OPEN Menu item");
    else
        // Must be the "Exit" menu item.
        System.exit(0);
}
```

The test code is on your CD-ROM. If you want to run it, type `java fancysrc .MenuEventTest`. At this point the menu looks right, and its events are being handled properly, so we can move on to the question of how to open a file.

Specifying a File

Most computer programs modify data stored in files. This accounts for the prominent position of the File menu. Selecting a file for a program to process is a very common activity. You would expect file selection to be standardized in some way, and this is indeed the case.

Java provides a class called `java.awt.FileDialog`, which supports all the functionality needed to help users specify a file. The class is easy to use. As the name implies, it creates a *dialog box*. A dialog box is a window that is subordinate to its program's main frame, used for brief user interaction. When you delete a file or exit a program, and a box pops up to ask you if you're sure, you are looking at a dialog box.

Many dialog boxes are *modal*. A modal dialog box consumes all mouse and keyboard input to the program. This implies that you can't continue using the program until you have dealt with the dialog box and dismissed it. Most "Are you sure?" dialog boxes are modal. Java's file dialog box is also modal.

The `FileDialog` class shares a lot of behavior with the `Frame` class. This is not surprising, since the classes have a common superclass called `Window`, as shown in [Figure 17.5](#).

A glance at the API shows that `FileDialog` class has three constructors:

```
FileDialog(Frame parent)
FileDialog(Frame parent, String title)
FileDialog(Frame parent, String title,
           int mode)
```

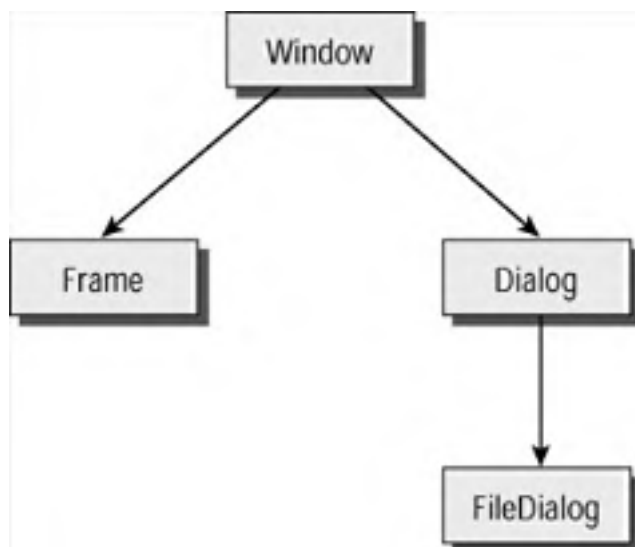


Figure 17.5: Window, Frame, and FileDialog

The `parent` argument is the frame over which the dialog box will appear. The `title` string determines what appears in the dialog box's title bar. The `mode` specifies whether the dialog box will be used for opening or saving a file. Opening is the default, so you don't have to worry about specifying the mode. (But see Exercise 1 at the end of this chapter.)

[Figure 17.6](#) shows a file dialog box, configured for opening:

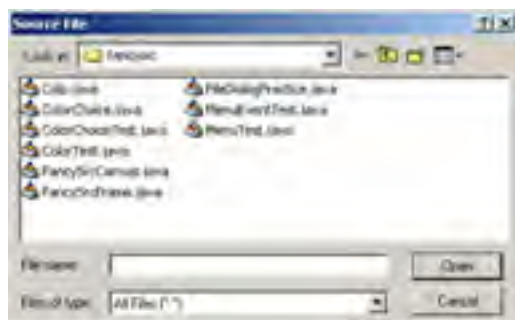


Figure 17.6: File dialog box configured for opening

Unlike frames, file dialog boxes are created with non-zero width and height, so you don't have to call `setSize()` on them. However, like frames, they are not visible until you call `setVisible(true)` on them. When you make this call, the dialog box appears, and the rest of the program's GUI refuses to accept mouse or keyboard input. Moreover, execution of your program pauses. Eventually, the user deals with and dismisses the dialog box. At this point, the rest of the GUI once more accepts input, and execution of your program continues from the line immediately following the `setVisible(true)` call.

The functionality is complicated, but using file dialog boxes is actually very simple. You construct your dialog box and, at the right moment, call `setVisible(true)` on it. The next line of code will not execute until a file has been specified (or the user has selected Cancel). There are two useful calls that you can then make on your dialog box, and both methods return strings:

`getFile()` The `getFile()` method returns the name of the file the user chose, or `null` if the dialog box was canceled.

`getDirectory()` The `getDirectory()` method returns the name of the chosen directory.

Whenever you learn about a new Java class, it's a good idea to write a practice program that creates an instance of the class and uses it in a way similar to the way you will later be using it in your program. That way you can experiment freely, and there is no danger that you will break your project accidentally by deleting or changing perfectly good code.

Here is a practice program that creates a file dialog box when a button is clicked:

```
package fancysrc;

import java.awt.*;
import java.awt.event.*;

class FileDialogPractice extends Frame implements ActionListener
{
    public FileDialogPractice()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Show me...");
        btn.addActionListener(this);
        add(btn);
        setSize(200, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        FileDialog dia = new FileDialog(this);
        dia.setVisible(true);
        String fileName = dia.getFile();
        if (fileName == null)
            System.out.println("You canceled the dialog.");
        else
            System.out.println("Your chose file " + fileName + "
                               in " + dia.getDirectory());
    }

    public static void main(String[] args)
    {
        (new FileDialogPractice()).setVisible(true);
    }
}
```

Notice the code in `actionPerformed()`:

1. `FileDialog dia = new FileDialog(this);`
2. `dia.setVisible(true);`
3. `String fileName = dia.getFile();`
- ...

After line 1 executes, processing does not move on to line 2 until the user has dismissed the dialog box. The `getFile()` call on line 3 returns `null` if the dialog box was canceled. If you want to try the test program, it's on your CD-ROM. To run it, type `java fancysrc.FileDialogPractice`.

What should the code do after the file has been specified? We don't know yet, but we will figure it out in good time. At this point, we have code to capture the user's desired input file. Before we worry about processing and displaying the file, let's turn our attention to the remaining GUI-related piece of the puzzle.

Specifying Colors

This section will look at the portion of the GUI that supports color selection. First you'll see a perfectly reasonable design: straightforward, but nothing fancy. Then the design will be improved in stages, ending with code that is elegant and reusable.

Let's start with what we know. We want to users to select from among a small number of colors. The colors must be dark enough that they can be read easily on a white background. That rules out yellow, pink, and several others. Let's settle on these:

- Black
- Blue
- Green
- Red

- Cyan
- Magenta

Cyan and magenta are marginal. For now, we'll include them. We can throw them out later on if they don't look good. If throwing them out proves to be difficult, that's an indication that our design wasn't very flexible.

Our users will have to select from these six colors... twice. Once for the keyword color, and once for the comment color. Of the components that you learned about in [Chapter 15](#), there are two that support making an exclusive selection from a small set of options: choices and radio buttons. We can rule out radio buttons because we would need 12 of them, compared to only two choices. If we used radio buttons, they would dominate the GUI, forcing the control area to be much larger than it needs to be, as shown in [Figure 17.7](#).



Figure 17.7: Too many radio buttons

For this situation, choices are much cleaner. Let's assume that our main application class will be called `FancySrcFrame` and will extend `Frame`. The class code will include the following declarations:

```
private Choice keywordChoice, commentChoice;
```

The choice components should be built in the `FancySrcFrame` constructor. One way to build them would be like this:

```
keywordChoice = new Choice();
keywordChoice.add("BLACK");
keywordChoice.add("BLUE");
keywordChoice.add("GREEN");
keywordChoice.add("RED");
keywordChoice.add("CYAN");
keywordChoice.add("MAGENTA");
commentChoice = new Choice();
commentChoice.add("BLUE");
commentChoice.add("BLACK");
commentChoice.add("GREEN");
commentChoice.add("RED");
commentChoice.add("CYAN");
commentChoice.add("MAGENTA");
```

This code can be improved, because every call appears twice. Whenever code is duplicated, consider the alternative of creating a method. The following code is much easier to read and more reliable:

```
keywordChoice = buildColorChoice();
commentChoice = buildColorChoice();
...
private Choice buildColorChoice()
{
    Choice c = new Choice();
    c.add("BLACK");
    c.add("BLUE");
    c.add("GREEN");
    c.add("RED");
    c.add("CYAN");
    c.add("MAGENTA");
    return c;
}
```

The new version is 13 lines long, compared to 14 in the original. That's not much of a difference, but later you might want to add a third color choice, and perhaps a fourth. In the old version, each additional color choice required seven lines, compared to only one line in the new version. Moreover, all choice components created by the `buildColorChoice()` method will be identical. With the original approach, each time you type the seven repeated lines, you introduce the possibility of a transcription error. Did you notice that in the first block of code, the second choice reverses the order of `BLACK` and `BLUE`?

An even cleaner approach uses an array of color names. The following would appear along with the other variables of the `FancySrcFrame` class:

```
private String[] colorNames =
{
    "BLACK", "BLUE", "GREEN", "RED", "CYAN", "MAGENTA"
};
```

Now the `buildColorChoice()` method is just the following:

```
private Choice buildColorChoice()
{
    Choice c = new Choice();
    for (int i=0; i<colorNames.length; i++)
        c.add(colorNames[i]);
    return c;
}
```

If you want to add or remove colors from the set of options, you just edit the contents of `colorNames`.

The choices will need an item listener. The logical candidate is the `FancySrcFrame` class. The `itemStateChanged()` method should cause the display to be repainted, using the new keyword or comment color. Somewhere (we don't need to decide where right now), some code will have to figure out which colors to use, based on the settings of the two choices. One way to do this would be to have a method that returns an instance of `Color`:

```
private Color getColorFromChoice(Choice c)
{
    int index = c.getSelectedIndex();
    if (index == 0)
        return Color.BLACK;
    else if (index == 1)
        return Color.BLUE;
    else if (index == 2)
        return Color.GREEN;
    else if (index == 3)
        return Color.RED;
    else if (index == 4)
        return Color.CYAN;
    else
        return Color.MAGENTA;
}
```

That certainly works, but there is a much cleaner way. First, we'll create an array of colors. For maximum readability, it should appear next to the `colorNames` array:

```
private Color[] colors =
{
    Color.BLACK, Color.BLUE, Color.GREEN,
    Color.RED, Color.CYAN, Color.MAGENTA
};
```

To determine the color indicated by a choice component, use the choice's selected index as an index into the `colors` array:

```
private Color getColorFromChoice(Choice c)
{
    int index = c.getSelectedIndex();
    return colors[index];
}
```

We have now worked out one piece of our design. We could go on to work out all our other design decisions, but before we blaze ahead, let's test what we have so far. If it doesn't work, we need to try again. If it works, we aren't committed to it. We reserve the right to improve on our color-specifying design later on.

Since color specification is the first code we will develop, our test will be simple. We don't yet know how we will select the file to be read, or paint lines on the screen, or paint source code on the screen in appropriate colors. So we'll create a program that just implements the color-specifying part of the GUI. To verify that the right colors are being returned from `getColorFromChoice()`, we'll just draw two squares in the frame. The square on the left will be the keyword color. The square on the right will be the comment color. [Figure 17.8](#) shows the GUI.

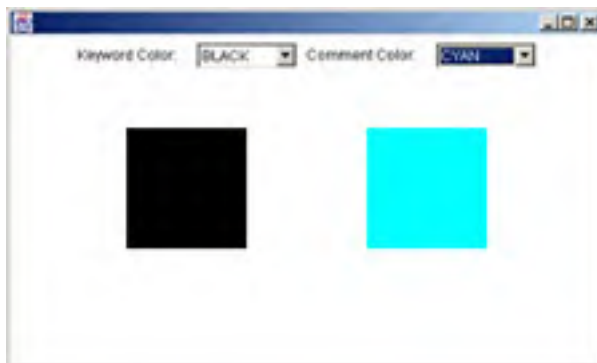


Figure 17.8: Testing color selection

Here's the code:

```
package fancysrc;

import java.awt.*;
import java.awt.event.*;

class ColorTest extends Frame implements ItemListener
{
    private String[] colorNames =
    {
        "BLACK",    "BLUE",    "GREEN",
        "RED",      "CYAN",   "MAGENTA"
    };

    private Color[] colors =
    {
        Color.BLACK, Color.BLUE, Color.GREEN,
        Color.RED,   Color.CYAN, Color.MAGENTA
    };
}
```

```
private Choice keywordChoice, commentChoice;

public ColorTest()
{
    setLayout(new FlowLayout());
    add(new Label("Keyword Color:"));
    keywordChoice = buildColorChoice();
    keywordChoice.addItemListener(this);
    add(keywordChoice);
    add(new Label("Comment Color:"));
    commentChoice = buildColorChoice();
    commentChoice.addItemListener(this);
    add(commentChoice);
    setSize(500, 300);
}

private Choice buildColorChoice()
{
    Choice c = new Choice();
    for (int i=0; i<colorNames.length; i++)
        c.add(colorNames[i]);
    return c;
}

private Color getColorFromChoice(Choice c)
{
    int index = c.getSelectedIndex();
    return colors[index];
}

public void itemStateChanged(ItemEvent e)
{
    repaint();
}

public void paint(Graphics g)
{
    Color keywordColor = getColorFromChoice(keywordChoice);
    g.setColor(keywordColor);
    g.fillRect(100, 100, 100, 100);
    Color commentColor = getColorFromChoice(commentChoice);
    g.setColor(commentColor);
    g.fillRect(300, 100, 100, 100);
}

public static void main(String[] args)
{
    new ColorTest().setVisible(true);
}
}
```

The class code begins with the arrays `colorNames` and `colors`. Notice how each name is aligned vertically with its corresponding color. It's a small touch that creates a visual relationship between the functionally related items.

The `itemStateChanged()` method just calls `repaint()`. Remember from [Chapter 16](#) that when you want to paint your display in reaction to user input, you shouldn't directly call `paint()`. Rather, you should call `repaint()`, which clears the display and then calls `paint()`. Our `paint()` method draws the two squares.

It works. If you want to try it, the code is on your CD-ROM. Just type `java fancysrc .ColorTest`.

We can't rest on our laurels yet. The code works, `ColorTest` proves it, but it isn't very object-oriented. The software that supports a single function (color selection) is spread throughout the class. The great thing about object-oriented programming is that it allows you to encapsulate related functionality. Let's see how to encapsulate color selection.

Think about the `Choice` class. Its `getSelectedIndex()` method returns an `int`. A lot of the code in `ColorTest` is devoted to converting that `int` to the corresponding color. Life would be a lot easier if `Choice` had a method called `getSelectedColor()`. Of course, no such method exists, because `Choice` is a general-purpose class intended for specifying colors, fonts, font sizes, names, countries, languages, or anything else that any programmer might think of. But we can subclass `Choice` to create a special-purpose class that does exactly what we want.

We will create a subclass called `ColorChoice`. The constructor will populate the component with the appropriate strings. The `colorNames` and `colors` arrays will go inside the new class, since no other code will need them. We will provide a `getSelectedColor()` method. The new class looks like this:

```
package fancysrc;

import java.awt.*;

public class ColorChoice extends Choice
{
    private static String[] colorNames =
    {
        "BLACK",      "BLUE",      "GREEN",
        "RED",         "CYAN",     "MAGENTA"
    };

    private static Color[] colors =
    {
```

```
        Color.BLACK, Color.BLUE, Color.GREEN,
        Color.RED,   Color.CYAN, Color.MAGENTA
    };

    public ColorChoice()
    {
        for (int i=0; i<colorNames.length; i++)
            add(colorNames[i]);
    }

    public Color getSelectedColor()
    {
        return colors[getSelectedIndex()];
    }
}
```

Notice that the two arrays have been declared as `static`. Remember that if a variable is static, there is only one copy of it, shared by all instances of the class. We know that there will be two instances of `ColorChoice`. There is no need to create two identical versions of the arrays, which is what would happen if they were not static. Each instance would have its own version. If the GUI changed later so that there were 25 color choices, there would be 25 identical versions of each array. Duplication of data is always something to be avoided. Here we avoid it by making the arrays static.

Testing the code is much easier. The complicated stuff is now in the `ColorChoice` class. The test code becomes the following:

```
package fancysrc;

import java.awt.*;
import java.awt.event.*;

class ColorChoiceTest extends Frame implements ItemListener
{
    private ColorChoice keywordChoice, commentChoice;

    public ColorChoiceTest()
    {
        setLayout(new FlowLayout());
        add(new Label("Keyword Color:"));
        keywordChoice = new ColorChoice();
        keywordChoice.addItemListener(this);
        add(keywordChoice);
        add(new Label("Comment Color:"));
        commentChoice = new ColorChoice();
        commentChoice.addItemListener(this);
        add(commentChoice);
        setSize(500, 300);
    }

    public void itemStateChanged(ItemEvent e)
    {
        repaint();
    }

    public void paint(Graphics g)
    {
        Color keywordColor = keywordChoice.getSelectedColor();
        g.setColor(keywordColor);
        g.fillRect(100, 100, 100, 100);
        Color commentColor = commentChoice.getSelectedColor();
        g.setColor(commentColor);
        g.fillRect(300, 100, 100, 100);
    }

    public static void main(String[] args)
    {
        new ColorChoiceTest().setVisible(true);
    }
}
```

The arrays are gone. The variables `keywordChoice` and `commentChoice` are now declared as type `ColorChoice`. We can call `addItemListener()` on them, just as if they were instances of `Choice`, because they inherit all the event-processing functionality of `Choice`. If you want to run the code, type `java fancysrc.ColorChoiceTest`. The GUI looks just like the earlier test GUI, so there's no need for a screenshot.

Now we can rest on our laurels! The source for class `ColorChoice` is less than 30 lines long, and look at what it can do:

- Look good.
- Behave exactly like a standard `Choice`.
- Be manipulated by a layout manager.
- Send out item events when activated.
- Report the selected color.

That's not a bad resume for such a small class. But we can't rest on our laurels all day. It's time to develop the rest of the code.

The Main Display Area

Most of the GUI work is now complete. We still have to create the check box that requests lines, but we'll get to that a bit later. Now we're going to step back and look at the big picture.

Our display will be involved a lot of painting. The painting examples you saw in [Chapter 14](#) all involved painting a frame. You saw subclasses of `java.awt.Frame` with specialized versions of the `paint()` method. Later you saw that when user input makes it necessary to revise the display, you should call your frame's `repaint()` method, which causes the screen to be cleared and the `paint()` method to be called.

As it happens, the `repaint()` mechanism works for certain other component types in addition to frames. There is a class called `java.awt.Canvas` that has no inherent appearance at all. If you construct a canvas and install it in a GUI, you won't see anything worth mentioning. That's okay, because you never actually put a canvas in a GUI. You create a subclass of `Canvas`, with a `paint()` method that draws whatever you want, and it is the subclass that you use in your GUI.

We will use a `Canvas` subclass, called `FancySrcCanvas`, for our main display area. The frame that contains everything will use its default `Border` layout manager. The control components (the Show lines check box and the two color choices) will go in a panel at North, and the canvas will be at Center, as shown in [Figure 17.9](#).



Figure 17.9: GUI layout

The `FancySrcCanvas` will need to redisplay itself whenever the user changes the file, the Show lines preference, or the keyword or comment color. The GUI code will detect all these changes. The `FancySrcCanvas` class needs a method that the GUI can call when it's time to redisplay. Let's call this method `reconfigure()`. It will need four arguments:

- A string representing the name of the new source file.
- A boolean that controls whether or not lines should be displayed.
- Colors for keywords.
- Colors for comments.

The method should not paint directly to the screen, because painting is always relegated to the `paint()` method. Our `reconfigure()` method will simply record its four arguments and then call `repaint()`. This will trigger a behind-the-scenes chain of events that will clear the canvas and call `paint()`. When `paint()` runs, it will know what to do (what file to read, what colors to use, whether it should underline), because it will read the values stored by `reconfigure()`.

We can now write the skeleton of `FancySrcCanvas`:

```
public class FancySrcCanvas extends Canvas
{
    private String    fileName;
    private boolean   showLines;
    private Color     keywordColor, commentColor;

    FancySrcCanvas()
    {
        // To do
    }

    void reconfigure(String file, boolean line,
                    Color kColor, Color cColor)
    {
        fileName = file;
        showLines = line;
        keywordColor = kColor;
        commentColor = cColor;
        repaint();
    }

    public void paint(Graphics g)
    {
        // To do
    }
}
```

The body of the constructor and the `paint()` method have been left for later. The constructor will be trivial, but `paint()`, as you

might expect, will be substantial.

Once again, as you saw with the `ColorChoice` class, subclassing allows us to create clean, encapsulated code. If we did not use a canvas subclass, the painting would happen in the `paint()` method of the main frame subclass. This painting code would be jumbled in along with all the other code. With subclassing, we know that all the painting code, and nothing except the painting code, is to be found in `FancySrcCanvas`.

Painting Colored Code

Now we have a workable concept for the `FancySrcCanvas` class, so we can fill in the details. The boring details go in the constructor. The interesting ones go in the `paint()` method.

Let's dispense with the boring details first. We need to choose a font and make some decisions about how to lay out the lines of text. The values we'll use here are somewhat arbitrary. We need to decide on a y coordinate for the topmost line of code. (Remember, when you paint text, you specify the text's baseline, not the top of the text). We need to decide how much vertical space to leave between consecutive lines of code, and we need to choose an x coordinate for the text. [Figure 17.10](#) shows how text will be positioned.

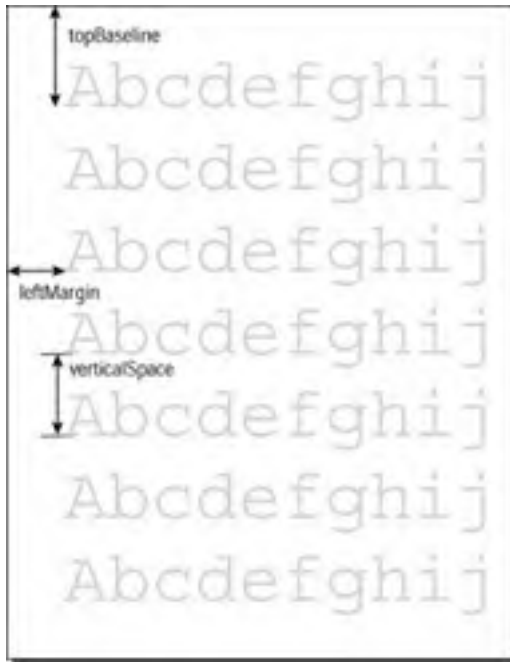


Figure 17.10: Positioning text

We'll use a plain monospaced 16-point font. (Remember, monospaced fonts are always best for displaying source code.) The topmost baseline will be at 20. Every line of text will be 18 pixels below the previous line. The x-coordinate of all text will be 9. These values were arrived at after a fair amount of boring experimentation. The result is the following constructor for

`FancySrcCanvas`:

```
FancySrcCanvas ()
{
    font = new Font("Monospaced", Font.PLAIN, 16);
    topBaseline = 20;
    leftMargin = 9;
    verticalSpace = 18;
}
```

With the font we have chosen, each character is 10 pixels wide. This number will be extremely useful later on. For now, can you guess why it's important?

There is a riddle that brings a knowing gleam to the eyes of experienced programmers, even though it isn't very funny. *How do you fit five elephants into a Volkswagen Beetle? Answer: two in the front, three in the back.* It isn't a good riddle, but it's a good example of top-down development, where you begin with an overall design idea, breaking each piece down into successively more refined designs until there is nothing left to do but implement your solution. Of course, if the design doesn't work, it's not the fault of the elephants.

We will take a top-down approach to developing the `paint()` method of `FancySrcCanvas`, starting with what we know. We know that the horizontal lines must be painted if `showLines` is `true`. We also know that the text must be painted if a source file has been specified. That is, if `fileName`, which is initialized to `null`, is no longer `null`. That's all we know, but it's enough to start. Here's our `paint()` method:

```
public void paint(Graphics g)
{
    if (showLines)
        paintLines(g);
    if (fileName != null)
        paintText(g);
}
```


Now we have to create the `paintLines()` and `paintText()` methods. `paintLines()` seems easy. Starting from the topmost baseline, horizontal lines must be drawn across the entire width of the canvas. Lines must be drawn `verticalSpace` pixels apart, down to the bottom of the canvas. It would all be simple, if only we knew how wide and tall the canvas is.

Fortunately, Canvas has a `getSize()` method that returns an instance of `java.awt.Dimension`. This very simple class has variables `width` and `height`. So the code can use `getSize().width` and `getSize().height` to determine the size of the canvas.

Here is the `paintLines()` code:

```
private void paintLines(Graphics g)
{
    g.setColor(Color.lightGray);
    int height = getSize().height;
    int width = getSize().width;
    for (int y=topBaseline; y<height; y+=verticalSpace)
        g.drawLine(0, y, width, y);
}
```

Now it's time to write `paintText()`. We don't yet know how we're going to draw text in three colors, but we don't have to know. This is top-down development. Let's stay with what we *do* know. There's a Java source file whose name is found in the variable `fileName`. We know that `paintText()` will need to read each line in turn from that file and paint the line. So here is the method, with the issue of painting multicolored text deferred for later consideration:

```
1. private void paintText(Graphics g)
2. {
3.     g.setFont(font);
4.
5.     try
6.     {
7.         // Create the readers.
8.         FileReader fr = new FileReader(fileName);
9.         LineNumberReader lnr = new LineNumberReader(fr);
10.
11.        // Read & display.
12.        String s = "xx"; // Anything but null
13.        int y = topBaseline;
14.        while (s != null)
15.        {
16.            s = lnr.readLine();
17.            if (s == null)
18.                break;
19.            paintOneSourceLine(g, s, y);
20.            y += verticalSpace;
21.        }
22.
23.        // Close the readers.
24.        lnr.close();
25.        fr.close();
26.    }
27.
28.    catch (IOException x)
29.    {
30.        System.out.println("Trouble!" + x.getMessage());
31.    }
32. }
```

The method uses a file reader chained to a line number reader. You were introduced to readers in [Chapter 13, "File Input and Output"](#). The `while` loop in lines 14-21 reads lines of text from the file until the `readLine()` call on line 16 returns `null`, indicating that the end of the file has been reached. (On line 12, `s` has to be initialized to any non-`null` string, so that the loop won't terminate the first time through.)

The variable `y` determines the baseline of the next line of text to be painted. On line 13, `y` is initialized to `topBaseline`. Every pass through the loop, it is incremented by `verticalSpace` (line 20).

Text is painted on line 19, where a call is made to `paintOneSourceLine()`. We'll write this method shortly. Its arguments are the `Graphics` object, the string to be painted (`s`), and the `y`-coordinate of the text (`y`).

We have deferred thinking about how to paint multicolored source text until we could create a good structure for the `paint()` method of `FancySrcCanvas`. That structure is now in place, so it's time to decide how to paint the code. Here's the skeleton of `paintOneSourceLine()`:

```
private void paintOneSourceLine(Graphics g,
                               String srcLine,
                               int y)
{
    ...
}
```

Here's the strategy. First, the method will paint the entire text line in black, whether or not it contains any keywords or comments. Then the line will be inspected to see if it contains any keywords or comments. If so, part of the text will be painted again, in the appropriate color. As you will see, there are methods in the `String` class that make this easy.

Let's start by painting the entire line in black. We don't have to call `setFont()` because that call was made already, in `paintText()`:

```
// First paint entire line in black.
g.setColor(Color.black);
g.drawString(srcLine, leftMargin, y);
```

That was easy. Now to detect and render comments. Comments begin with a double slash (//) and continue through the end of the line. So the code needs to answer the following questions:

Does the string contain a double slash?

If so, where is the double slash?

If the answer to the first question is "no," there is no comment to paint.

Fortunately, class `String` has a method called `indexOf()`. Its argument is another string. If the argument string appears anywhere in the executing object string, the method returns the position of the argument string within the executing object string. For example, if `s1` is `whether` and `s2` is `the`, `s1.indexOf(s2)` is 3. If the argument string does not appear in the executing object string, `indexOf()` returns -1. For example, if `s1` is `whether` and `s2` is `heather`, `s1.indexOf(s2)` is -1.

The comment-painting code uses another method of the `String` class: `substring()`. You were introduced to this method in [Chapter 12, "The Core Java Packages and Classes."](#) When called with a single `int` argument, it returns the portion of the string beginning at the argument position. For example, if `s1` is `whether`, `s1.substring(2)` is `ether`.

Here is the code that paints comments:

```
1. // Paint comment (if any).
2. int commentIndex = srcLine.indexOf("//");
3. if (commentIndex >= 0)
4. {
5.     g.setColor(commentColor);
6.     String comment = srcLine.substring(commentIndex);
7.     int x = charIndexToX(commentIndex);
8.     g.drawString(comment, x, y);
9. }
```

To illustrate how this code works, consider what happens when `srcLine` is
`height += 25; // Increment height`

The comment begins at character position 14, so `commentIndex` is 14. On line 6, `comment` is `//increment height`. This is the string that is overpainted in the comment color, at line 8.

Line 7 makes a call to `charIndexToX()`, which returns the x-coordinate where the comment will be painted. This value must be calculated exactly, so that the new text will exactly overwrite the black text. This method is

```
private int charIndexToX(int charIndex)
{
    return leftMargin + 10*charIndex;
}
```

Earlier in this section, you read that in a 16-point monospaced font, each char is 10 pixels wide. This implies that, for example, the 18th character in any line is 180 pixels to the right of the 0th character. And the 0th character is always painted at `leftMargin`. So the x-coordinate of the `n`th character is `leftMargin + 10*n`. This is the formula used by `charIndexToX()`.

So far our `paintOneSourceLine()` code is

```
private void paintOneSourceLine(Graphics g,
                               String srcLine,
                               int y)
{
    // First paint entire line in black.
    g.setColor(Color.black);
    g.drawString(srcLine, leftMargin, y);

    // Paint comment (if any).
    int commentIndex = srcLine.indexOf("//");
    if (commentIndex >= 0)
    {
        g.setColor(commentColor);
        String comment = srcLine.substring(commentIndex);
        int x = charIndexToX(commentIndex);
        g.drawString(comment, x, y);
    }
}
```

We are ready to deal with keywords, but we have to be careful. If we just search for keywords and overwrite them in the right color, we could get confounded by a line like this:

```
x = 16; // try to while away the time
```

The code contains no Java keywords, but the comment does. When the line is being searched for keywords, the search should not include the comment. This will not guarantee that the code will never erroneously color non-keyword text, but it guards against one common situation. (Making the keyword search 100% foolproof would be a daunting task. It would overwhelm the code and would seriously reduce the learning value of the project. Not searching comments will be enough for our purposes. Exercise 5 at the end of this chapter invites you to think more about the problem.)

If the software is going to search for keywords, it needs to know which strings are keywords. The `FancySrcCanvas` class needs an array of strings that are Java keywords. Here it is:

```
private String[] keywords =
{
    "abstract", "boolean", "break", "byte", "case", "catch",
    "char", "class", "continue", "default", "double", "do",
    "else", "extends", "false", "final", "float", "for",
    "if", "implements", "import", "instanceof", "int",
    "interface", "long", "new", "null", "package", "private",
    "protected", "public", "return", "short", "static",
    "super", "switch", "this", "throws", "throw", "true",
    "try", "void", "while"
};
```

Actually, the list is incomplete. It only includes Java keywords that were introduced in this book. There are a handful of others. Strictly speaking, `null`, `true`, and `false` are not keywords, but something similar.

Here is the skeleton of the remainder of the `paintOneSourceLine()` code:

```
// Search every position in string, through comment,
// for any keyword.
g.setColor(keywordColor);
int lastCharPosition = srcLine.length()-1;
if (commentIndex >= 0)
    lastCharPosition = commentIndex - 1;
for (int index=0; index<=lastCharPosition; index++)
{
    ...
}
```

The `for` loop will search every position in the line of code, through `lastCharPosition`, to see if it begins with any entry in the `keywords` array. If the line does not contain a double-slash comment, `lastCharPosition` is set to the last character position in the line. If a double-slash comment is present, `lastCharPosition` is set to the last character position before the comment. For example, suppose the source line is

```
x = new Line();// Construct a line
```

The `for` loop will check each of the following substrings:

```
x = new Line();
 = new Line();
= new Line();
 new Line();
new Line();
ew Line();
w Line();
 Line();
Line();
ine();
ne();
e();
();
);
;
```

Each of the substrings will be compared against each entry in the `keywords` array. The code will use two methods of `String` that were presented in [Chapter 12](#), `substring()` and `startsWith()`. The `substring()` method takes an `int` argument. It returns the portion of the original string beginning at the specified index. For example, if `s1` is `Meryl Streep`, `s1.substring(9)` is `eep`. The `startsWith()` method takes a string argument. It returns `true` if the original string starts with the argument string. For example, if `s1` is `Meryl Streep` and `s2` is `Me`, `s1.startsWith(s2)` is `true`.

Now the body of the `for` loop can be filled in:

```
1. for (int index=0; index<=lastCharPosition; index++)
2. {
3.     // Search at this position for every keyword.
4.     String sub = srcLine.substring(index);
5.     for (int i=0; i<keywords.length; i++)
6.     {
7.         if (sub.startsWith(keywords[i]))
8.         {
9.             int x = charIndexToX(index);
10.            g.drawString(keywords[i], x, y);
11.            break; // Can't be any more keywords here
12.        }
13.    }
14. }
```

Recall that the `Graphics` object has already had its color set to the keyword color. Line 9 makes use of the `charIndexToX()` method, which was written for the comment-painting code, to compute where to overdraw the keyword string.

That's all for the `FancySrcCanvas` class. Here is the whole class listing, all in one place:

```
package fancysrc;

import java.io.*;
import java.awt.*;

class FancySrcCanvas extends Canvas
{
    private String[] keywords =
    {
        "abstract", "boolean", "break", "byte", "case", "catch",
        "char", "class", "continue", "default", "do", "double",
        "else", "extends", "false", "final", "float", "for",
        "if", "implements", "import", "instanceof", "int",
        "interface", "long", "new", "null", "package", "private",
        "protected", "public", "return", "short", "static",
        "super", "switch", "this", "throws", "throw", "true",
        "try", "void", "while"
    };

    private Font          font;
    private int           topBaseline;
    private int           leftMargin;
    private int           verticalSpace;
    private String        fileName;
    private boolean       showLines;
    private Color         keywordColor, commentColor;

    FancySrcCanvas()
    {
        font = new Font("Monospaced", Font.PLAIN, 16);
        topBaseline = 20;
        leftMargin = 9;
        verticalSpace = 18;
    }

    void reconfigure(String file, boolean line,
                     Color kColor, Color cColor)
    {
        fileName = file;
        showLines = line;
        keywordColor = kColor;
        commentColor = cColor;
        repaint();
    }

    public void paint(Graphics g)
    {
        if (fileName == null)
            return;

        if (showLines)
            paintLines(g);

        paintText(g);
    }

    private void paintLines(Graphics g)
    {
        g.setColor(Color.lightGray);
        int height = getSize().height;
        int width = getSize().width;
        for (int y=topBaseline; y<height; y+=verticalSpace)
            g.drawLine(0, y, width, y);
    }

    private void paintText(Graphics g)
    {
        g.setFont(font);

        try
        {
            // Create the readers.
            FileReader fr = new FileReader(fileName);
            LineNumberReader lnr = new LineNumberReader(fr);

            // Read & display.
            String s = ""; // Anything but null
            int y = topBaseline;
            while (s != null)
            {
                s = lnr.readLine();
                if (s == null)
                    break;
                if (s.startsWith("import ") || s.startsWith("package "))
                    g.setColor(keywordColor);
                else if (s.startsWith("//"))
                    g.setColor(commentColor);
                else
                    g.setColor(Color.black);
                g.drawString(s, leftMargin, y);
                y += verticalSpace;
            }
        }
        catch (IOException e)
        {
            // Ignore
        }
    }
}
```

```
        break;
        paintOneSourceLine(g, s, y);
        y += verticalSpace;
    }

    // Close the readers.
    lnr.close();
    fr.close();
}

catch (IOException x)
{
    System.out.println("Trouble! " + x.getMessage());
}
}

private void paintOneSourceLine(Graphics g,
                                String srcLine, int y)
{
    // First paint entire line in black.
    g.setColor(Color.black);
    g.drawString(srcLine, leftMargin, y);

    // Paint comment (if any).
    int commentIndex = srcLine.indexOf("//");
    if (commentIndex >= 0)
    {
        g.setColor(commentColor);
        String comment = srcLine.substring(commentIndex);
        int x = charIndexToX(commentIndex);
        g.drawString(comment, x, y);
    }

    // Search every position in string, through comment,
    // for any keyword.
    g.setColor(keywordColor);
    int lastCharPosition = srcLine.length();
    if (commentIndex >= 0)
        lastCharPosition = commentIndex - 1;
    for (int index=0; index<=lastCharPosition; index++)
    {
        // Search at this position for every keyword.
        String sub = srcLine.substring(index);
        for (int i=0; i<keywords.length; i++)
        {
            if (sub.startsWith(keywords[i]))
            {
                int x = charIndexToX(index);
                g.drawString(keywords[i], x, y);
                break; // Can't be any more keywords here
            }
        }
    }
}

private int charIndexToX(int charIndex)
{
    return leftMargin + 10*charIndex;
}
}
```

Putting It All Together

The [previous section](#) presented the entire listing for `FancySrcCanvas`, which is the longest of the project's three source files. You already saw `ColorChoice` in the "Specifying Colors" Section. That leaves only `FancySrcFrame`, which follows. You have seen its important pieces, but they were *in pieces*. Here it is, all in one place:

```
package fancysrc;

import java.awt.*;
import java.awt.event.*;

public class FancySrcFrame extends Frame implements
    ActionListener, ItemListener
{
    private MenuItem      openMI, exitMI;
    private String        fileName;
    private Checkbox      showLinesBox;
    private ColorChoice   keywordChoice, commentChoice;
    private FancySrcCanvas srcCanvas;
    private FileDialog    dialog;

    FancySrcFrame()
    {
        // Build menu.
        MenuBar mbar = new MenuBar();
        Menu fileMenu = new Menu("File");
        openMI = new MenuItem("Open...");
        openMI.addActionListener(this);
        fileMenu.add(openMI);
        exitMI = new MenuItem("Exit");
        exitMI.addActionListener(this);
        fileMenu.add(exitMI);
        mbar.add(fileMenu);
        setMenuBar(mbar);

        // Build control panel.
        Panel panel = new Panel(); // Uses flow layout
        showLinesBox = new Checkbox("Show lines");
        showLinesBox.addItemListener(this);
        panel.add(showLinesBox);
        keywordChoice = new ColorChoice();
        keywordChoice.addItemListener(this);
        keywordChoice.select(1);
        panel.add(new Label("Keyword color"));
        panel.add(keywordChoice);
        commentChoice = new ColorChoice();
        commentChoice.addItemListener(this);
        commentChoice.select(2);
        panel.add(new Label("Comment color"));
        panel.add(commentChoice);
        add(panel, BorderLayout.NORTH);

        // Build text display panel.
        srcCanvas = new FancySrcCanvas();
        add(srcCanvas, BorderLayout.CENTER);

        // Set to a reasonable size.
        setSize(720, 550);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == openMI)
        {
            if (dialog == null)
                dialog = new FileDialog(this, "Source File",
                    FileDialog.LOAD);
            dialog.setVisible(true); // Modal
            if (dialog.getFile() == null)
                return; // Canceled
            fileName = dialog.getDirectory() + dialog.getFile();
            boolean underline = showLinesBox.getState();
            Color keywordColor =
                keywordChoice.getSelectedColor();
            Color commentColor =
```

```
        commentChoice.getSelectedColor();
        srcCanvas.reconfigure(fileName, underline,
            keywordColor, commentColor);
    }

    else // Must be "Exit" menu item
        System.exit(0);
}

// Called if user activity in checkbox or
// either choice.
public void itemStateChanged(ItemEvent e)
{
    boolean underline = showLinesBox.getState();
    Color keywordColor = keywordChoice.getSelectedColor();
    Color commentColor = commentChoice.getSelectedColor();
    srcCanvas.reconfigure(fileName, underline,
        keywordColor, commentColor);
}

public static void main(String[] args)
{
    (new FancySrcFrame()).setVisible(true);
}
}
```

The only part of this code that has not been explained already is the `itemStateChanged()` method. This is called when the user checks the Show lines check box or either color choice. There is no need to call `getSource()` and figure out which component caused the method call, because the response is the same in any case: A call is made to the `reconfigure()` method of the `FancySrcCanvas`.

Note The complete source to this project is on your CD-ROM, in the `FinalProjectSource` directory.

Team LIB

◀ PREVIOUS NEXT ▶

Goodbye! Don't Forget to Write!

Please take a moment to appreciate what happened in this chapter. You observed the development of a multisource application involving several hundred lines of code:

- The program uses `awt` components for input and paints graphics for its output.
- Each of the three source modules defines a subclass.
- One of the classes implements not just one interface, but two.
- There are loops and conditional statements, and an array.
- There are calls to methods that throw exceptions.

And it all made sense. You saw how it fit together. You deserve sincere congratulations for the work you had to do in [Chapters 1 through 16](#). Without that foundation, this chapter would make no sense at all. You are well on your way toward mastery of the Java language.

This book is by no means a complete introduction. There is a lot more to be said about the language, the core classes, and programming techniques. With your strong foundation, you're now qualified to learn it all.

Were the animated illustrations beneficial? As far as I know, and as far as anybody at Sybex knows, this is the first computer book to use them. If you have any suggestions for how they can be improved, or ideas for new ones, please e-mail them to groundupjava@sgsware.com.

Are you interested in learning more about Java? Would you like to see another volume, picking up where this one leaves off, also based on animated illustrations? Please write.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Write a program that creates a frame with a File menu. The menu should have two items, Save... and Exit. When Save... is selected, the code should display a file dialog box, configured for saving a file. When the user has specified a file via the dialog box, your code should output the name of the file. All the information you need is on the API page for `java.awt.FileDialog`.
2. The `FileDialog` class has a `setDirectory()` method that controls which directory the dialog box will display. Look up the method description in the API to become familiar with how it works. Modify the final project code so that when the file dialog box appears, it displays one of the directories on your computer where you have stored some of your own Java source code. This will make it easier to display your own work.
3. Write an application that displays a canvas subclass in a frame, at Center. The frame does not contain any other components.

Use the following code as the `paint()` method for the canvas subclass:

```
1. public void paint(Graphics G)
2. {
3.     g.setFont(new Font("Serif", Font.PLAIN, 24));
4.     g.setColor(Color.blue);
5.     g.drawString("Look at this!", 0, 0);
6. }
```

Run the program. Do you see what you expected to see? How do you explain the results?

Now change line 5 to this:

```
g.drawString("A bluejay in a quagmire", 0, 0);
```

Now do you see what you expected to see? Again, how do you explain the results?

4. The `FancySrcCanvas` class has an array of Java keywords. In that array, `throws` comes before `throw`. Otherwise, the list is alphabetical. Why does `throws` come before `throw`?
5. There are several situations in which the project code would improperly draw text in the keyword color. How many of these situations can you name?
6. How would you modify the project code so that `null`, `true`, and `false` are not rendered in the keyword color?

Appendix A: Downloading and Installing Java

This is the most important part of this book. Java is not a spectator sport. The best way to learn and enjoy it is to use it. Every chapter of this book except [Chapter 1](#) has programming assignments. You can't do them if your computer doesn't have Java. Equally important, the animated illustrations are all Java programs, and you can't run them without Java. So take the time right now to download and install it. You will be richly rewarded for your effort. Java is an educational tool that will keep you fascinated for the rest of your life.

Overview of the Process

We'll start these instructions with a brief overview of what is involved in downloading and installing. Then you should follow whichever one of the brand-specific sections corresponds to your own computer.

You're going to go to a Javasoft Web page and download two very large files (tens of megabytes). If you have fast Internet access, congratulations. If you don't, each download could take several hours. If that's the case, consider starting one download at the end of the day. The file will be there for you in the morning. That evening, do the same with the second file.

The first file is Java itself. It is an archive containing the Java compiler, the Java Virtual Machine, and various other helpful programs. (If you don't know what a compiler or Java Virtual Machine are, they're discussed in [Chapter 2](#).) The official name for this download is the SDK, or Software Developer's Kit. The second file contains the API pages, a huge collection of HTML pages that describe the core Java packages and classes. You won't need the API pages until [Chapter 12](#), but you might as well download them as soon as possible.

Before you run any Java program (including the compiler, which is itself a Java program), you have to add the location of Java's executables to your `PATH` environment variable. You may also need to set the `CLASSPATH` environment variable. There are many ways to set these values. The approach presented here involves creating a script to be run manually when you are ready to view the animated illustrations or play with Java. Avoid modifying boot-time or login scripts, because a small typing error can get you into a lot of trouble. Also, manual scripts are easier to undo.

The following section is about installing Java in Windows. If you use a Macintosh, please skip to the "[Macintosh](#)" section later in this appendix.

Windows

This section will walk you through downloading and installing Java on Windows-based computers.

You download Java for Windows from <http://java.sun.com/j2se/1.4/download.html>. This page contains a list of platforms for which the current version of Java is available. For most of these platforms, you can download a JRE (Java Runtime Environment) or an SDK (Software Developer's Kit). *You want the SDK, not the JRE.* At or near the top of the list, you will find Windows (U.S. English only) and Windows (all languages, including English). Click on the link for Windows (all languages, including English).

This opens a page for specifying optional personal information. Then you move on to a license agreement page. If you do not accept the license, you will not be allowed to proceed. The next page allows you to download the file you want. The file has a complicated name, something like `j2sdk-1_4_1_02-windows-i586.exe`.

Click on the link to start the download. If your browser asks you to choose between running the program from its current location or saving it to disk, save it to disk. You can save it anywhere you like, and you can delete it after you run it. You might as well save it in the `\` directory of your C disk.

Execute the file by clicking on its icon in Windows Explorer. The installation wizard begins by asking you to accept the license policy (again). Then you're asked where you want the Java files to be placed. Put them in the `\` directory of your C disk. We'll assume that you use the default. The default name is something complicated, like `j2sdk1.4.1.02`.

Next, you're asked to choose which parts of installation you want. You must select the program files. You don't need any of the other parts, but if you have the disk space, you might be interested in the demos.

Then the installation wizard finishes its work. When it's done, your disk contains a new structure that looks something like [Figure A.1](#).

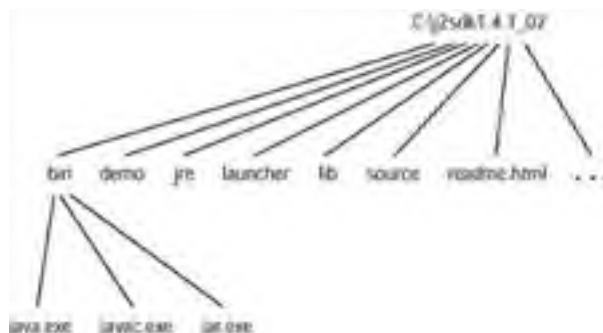


Figure A.1: Windows SDK file layout

The figure shows the three most important files in the `bin` subdirectory: the Java Virtual Machine (`java.exe`), the Java compiler (`javac.exe`), and the `jar` archive tool (`jar.exe`). Throughout this book, you will need the Java Virtual Machine and the compiler. You may need the archive tool shortly.

All the Java files are in place. Now create a directory where you will write Java programs. Again, you can call it whatever you want and put it wherever you like, but simpler is better. Here we'll assume your programming directory is called `C:\MyJavaCode`.

The next step is to create a batch file for setting your `PATH` and `CLASSPATH` environment variables. The `PATH` variable tells the operating system where to look for executable files when you run a program from the command line of a Command Prompt window. (And that is how you will run the animated illustrations and compile and execute your own programs.) The `CLASSPATH` variable tells Java where to look for class files. (That won't make sense unless you've read [Chapter 8](#).) You can call your script anything you like, and you can store it anywhere you like. The whole point of a batch file is to simplify things and reduce typing, so call it something simple and save it somewhere easy to remember. We will call it `\ja.bat`. It looks like this:

```
SET PATH=%PATH%;C:\j2sdk1.4.1.02\bin
SET CLASSPATH=.;D:\AnimatedIllustrations
CD C:\MyJavaCode
```

You don't have to use exactly this script, but if you like it there's a copy on the CD-ROM, in the `ExampleScripts\Windows` subdirectory. If you want to use it as-is, just copy it to the root directory of your primary disk drive. You might want to copy it anyway and use a text editor such as `Notepad` to edit it, rather than typing in your version from scratch.

The first line of the script appends Java's `bin` directory to your `PATH` environment variable, so that you will be able to run programs like `java`, `javac`, and `jar`. Remember that you will be running them by typing command lines into a Command Prompt window, not by double-clicking on icons.

The second line sets the `CLASSPATH` environment variable to the current directory, plus the directory on this book's CD-ROM where the animated illustration programs are stored. (If your CD-ROM drive letter is something other than `D`, substitute the appropriate letter. Alternately, you can copy the `AnimatedIllustrations` directory to your hard drive. That way you don't have to make sure the `CD-ROM` is loaded whenever you want to run an animated illustration. If you copy the `AnimatedIllustrations` directory to your hard drive, replace `D:\ AnimatedIllustrations` in the script with the full path, including drive letter, of the copied version of the `AnimatedIllustrations` directory.)

The third line of the script takes you into the directory where you will create and store the Java programs that you will write, in answer to some of the questions at the end of each chapter.

To test your work, open a Command Prompt window. Run your batch file script by typing `\ja`. The script runs. Note that it only affects the one Command Prompt window you are working in. If you close that window, you will have to open another one and then run the batch file again.

To make sure your script ran properly, type `java -version`. You get a message that tells you which version of Java is running, along with some other obscure, cryptic information that probably means something important to someone. This message means that you have installed Java and your `PATH` variable is set correctly. If the command doesn't work, make sure the full pathname of the `bin` directory in your script is spelled correctly, and that the directory contains `java.exe`.

Now type `java welcome.Welcome`. You should see a simple welcoming screen. This means that your `CLASSPATH` variable is set correctly. If you don't see the welcoming screen, make sure the value assigned to `CLASSPATH` in your script is spelled correctly, and that the directory is the `AnimatedIllustrations` directory from the CD-ROM. If you are running the animated illustrations directly from the CD-ROM (that is, if you didn't copy the files to your hard drive), make sure the CD-ROM is in the correct drive.

That takes care of the Java program files. They are all you need until [Chapter 12](#), where you will also need the API pages. These are a huge number of HTML pages to be viewed with the Web browser of your choice. You download them as a single zip file that you will have to extract.

You begin the download at the same page you visited before: <http://java.sun.com/j2se/1.4/download.html>. This page contains a list of several dozen products that you can download. Near the end of the list, you see J2SE 1.4.1 Documentation. Click on this item's Download link. After accepting another license agreement, you see a page with a link for downloading `j2sdk-1_4_1-doc.zip`. Click on the link. You're prompted to specify where you want to put the zip file. Put it in the directory where you stored your Java files. We recommend `C:\j2sdk1.4.1_02`.

Now you might have to extract the zip file. Some versions of Windows present a zip file as if it were a directory, extracting files only as needed. This saves a lot of space. If your system does this for you and it's satisfactory, you're done. To find out, open a Windows Explorer window and have it display your `C:\j2sdk1.4.1_02` directory. If `j2sdk-1_4_1-doc.zip` looks like a directory rather than a single file, you don't need to extract it if you don't want to. Otherwise, you need to extract.

To extract, you could double-click on the `j2sdk-1_4_1-doc.zip` icon and use Winzip to unpack the archive. But you might not have Winzip. Besides, there is a slicker way. If you haven't done so already, run your batch file script by typing `\ja`. Next, type `cd` into the directory that contains the `j2sdk-1_4_1-doc.zip` file. Unless you have done your own thing, the command to do so is `cd C:\j2sdk1.4.1_02`

Now type the following command:

```
jar xvf j2sdk-1_4_1-doc.zip
```

The `jar` command is one of the useful Java executables. It became usable when you ran the script and added `C:\j2sdk1.4.1_02\bin` to your path. Jar stands for *Java Archive*. It is like Winzip, but you run it from the command line. Fortunately, the `jar` file format is compatible with the `.zip` format, so you can use `jar` to extract any `.zip` archive.

Creating Program Files

Congratulations! You are ready to go. Right now you can run any animated illustration in the book. And please do... they are an essential part of your *Ground-Up Java* experience.

The other essential part of your experience is writing your own Java programs. You will do this when you work on the "write-a-program" questions at the end of the chapters, and of course you can write programs that implement your own ideas. We have already recommended that you do this in a directory called `\MyJavaCode`, and have your script "cd" in that directory. But now the question is, how do you create Java code? [Chapter 2](#) explained that writing a Java program means creating one or more files, called source files, in plain text format, with names that end with `.java`.

There are two ways to create Java source files:

- Use a general-purpose editor.
- Use an Integrated Development Environment (IDE).

You have several general-purpose editors installed on your system, including Notepad and Wordpad. It's a good idea to start with one of these; they are good enough for small programs. Use a fixed-width font like Courier to make your code line up nicely.

As you will learn from experience, you don't just write a program. The development process is an ongoing cycle of writing, testing, and modifying. So when you think you have finished writing your program, don't close your editing window. Leave it around, because in all likelihood you will want to make modifications or fix bugs. (Yes, this could happen even to you.)

After a while, you might get a vague sense that life could be better somehow. You might be ready for an IDE, or Integrated Development Environment. IDEs are products that help you create, maintain, debug, and keep track of Java programs. Many common operations (such as compiling) are achieved with a single button click. There are lots of IDEs on the market, ranging in price from free to expensive. It would be inappropriate to recommend one here, but if you type **Java + IDE** into your favorite Web search engine, you will get plenty of information.

A good IDE is a good thing, and a great IDE will greatly enhance your productivity. But a word of warning: Your goal right now is *not* to create large Java programs efficiently. Your goal is to learn as much as possible about Java. IDEs shield you from repetitive tasks. Before you start using them, it's a good idea to spend some time learning all the ins and outs and

details of Java, so that you'll know what the IDE is shielding you *from*. Spend some time with a general-purpose editor before you move on to an IDE.

The rest of this appendix is about installing Java on Macintosh. If you only have a Windows PC, you can skip the rest. Have fun!

Macintosh

This section will walk you through downloading and installing Java on Macintosh-based computers.

You download Java for Macintosh directly from Apple Computers. You will need three separate pieces:

- Mac OSX Developer Tools
- Java 1.4.1 Developer Tools Update
- Java 1.4.1. for Mac OS X and QTJava

First you will need to register as an Apple developer. This is free (as are the downloads), but you must do it before you can access the download sites. Just type this into your browser:

```
http://connect.apple.com
```

This will take you to the Apple Developer Connection site. From here, you can log in and download the software you want. If you are not yet a member, you must become one. Click the Join ADC button on the left side of the page, and answer the questions on the form. You're granted a user name and password to log in to the download site.

Once you've obtained your membership, go ahead and log in (same site as above). On the resulting page, click the Download Software link. You see a list of software packages along the left side of the page. Click on the Mac OS X link, and you see a list of possible items for download. Click the Download button immediately to the right of Dec 2002 Mac OS X Developer Tools. Downloading commences immediately. This file, `Dec2002DevToolsCD.dmg`, is 301.2MB and takes a little over an hour to download over a DSL line. Make a note of where you store the file on your local hard drive.

Now you need to update your Developer Tools. Use the Back button on your browser to return to the list of downloadable items. This time, choose Java. Click the Download button immediately to the right of Java 1.4.1 Developer Tools Update. Downloading commences immediately. This file, `Java141Developer.dmg`, is 48.6MB and takes about 20 minutes to download over a DSL line. Again, make a note of where you store the file on your local hard drive.

Lastly, you need to get the most recent version of Java for the Mac (1.4.1). Use the Back button to return to the list of downloadable items. Choose Java again. Click the Download button immediately to the right of Java 1.4.1 Update DP102. Downloading commences immediately. This file, `Java141Update1DP102.dmg`, is 37.4MB and takes about 15 minutes to download over a DSL line. Make a note of where you store the file on your local hard drive.

Now it's time to unpack and install the three files you've downloaded. First, the developer tools. Use Finder to navigate to the folder where you've placed the downloads. Locate the file `Dec2002DevToolsCD.dmg`, and double-click on its icon. You still need to be connected to the network during this process, because the `.dmg` file will attempt to mount the disk image of the Developer's Tools package for subsequent installation. You also need administrator-level permission to complete the installation. If all goes well, a small window labeled December 2002 Dev Tools appears. Double-click on the Developer icon and proceed with the installation as directed. When the installer asks you for a destination, select the default, Normal. When the installation has finished, you must reboot your computer before proceeding to the next installation.

Next it's time to unpack and install the Java update. Use Finder to navigate to the folder where you've placed the downloads. Locate the file `Java141Update1DP102.dmg` and double-click on its icon. A small window labeled Java 1.4.1 Update 1 appears. Double-click on the `Java1.4.1Update1.pkg` icon and proceed with the installation as directed. When the installer asks you for a destination, select the default, Normal. When the update has been unpacked and installed, you're directed to restart your computer.

Now it's time to unpack and install the Developer Tools update. Use Finder to navigate to the folder where you've placed the downloads. Locate the file `Java141Developer.dmg` and double-click on its icon. A small window labeled Java 1.4.1 Developer Update appears. Double-click on the `Java1.4.1Developer.mpkg` icon and proceed with the installation as directed. When the installer asks you for a destination, select the default, Normal.

Now all the Java files are in place. Create a directory where you will write Java programs. You can call it whatever you want and put it wherever you like, but simpler is better. Here we'll assume your programming directory is called `~/MyJavaCode`.

The next step is to add two lines to your `.login` or `.cshrc` file for setting your `path`:

```
set path=(lib:/usr/local/bin:/usr/ucb:/bin:/sbin:/usr/bin:/usr/sbin:/usr/etc:/Developer/Tools)
setenv CLASSPATH ./AnimatedIllustrations
alias gotoJava 'cd ~/MyJavaCode'
```

The first line tells the operating system where to look for executable files when you run a program from the command line of a terminal window. That's how you will run the animated illustrations, and compile and execute your own programs.

The second line sets the `CLASSPATH` environment variable to the current directory, plus the directory on this book's CD-ROM where the animated illustration programs are stored. The script assumes you have copied the entire CD-ROM to a directory on your hard drive called `/AnimatedIllustrations`. Doing so will make life simpler, because you can run the illustrations without the CD-ROM. If you don't want to copy the CD-ROM to your hard drive, or if you want to copy it to a different directory, modify the script accordingly.

The third line of the script is an alias that will take you to your Java work directory.

To test your work, log out and then log in again so that the script will execute. Open a terminal window (Applications/Utilities/Terminal in the Finder). To make sure your script ran properly, type `java -version`. You get a message that tells you which version of Java is running, along with some other obscure, cryptic information that probably means something important to someone. If you get this message, you have installed Java and your `PATH` variable is set correctly. If the command doesn't work, make sure the full pathname of the `bin` directory in your script is spelled correctly, and that the directory contains `java`.

Now type `java welcome.Welcome`. You should see a simple welcoming screen. This means that your `CLASSPATH` variable is set correctly. If you don't see the welcoming screen, make sure the value assigned to `CLASSPATH` in your script is spelled correctly, and that the directory is the `AnimatedIllustrations` directory from the CD-ROM. If you are running the animated illustrations directly from the CD-ROM (that is, if you didn't copy the files to your hard drive), make sure the CD-ROM is in the correct drive.

That takes care of the Java program files. They are all you need until [Chapter 12](#), where you will also need the API pages. These are a huge number of HTML pages to be viewed with the Web browser of your choice. You download them as a single zip file that you will have to extract.

You begin the download at <http://java.sun.com/j2se/1.4.1/download.html>. The page contains a list of several dozen products that you can download. Near the end of the list is J2SE 1.4.1 Documentation. Click on this item's Download link. After accepting another license agreement, you come to a page with a link for downloading `j2sdk-1_4_1-doc.zip`. Click on the link. You are prompted to specify where you want to put the zip file. Put it in a directory where you can find it easily. We recommend `~/MyJavaFiles`.

To extract, use `jar` to unzip the documents in the `j2sdk-1_4_1-doc.zip` file. If you haven't done so already, run your batch file script by typing `gotoJava`. Unless you have done your own thing, the command to unzip the file is

```
jar xvf j2sdk-1_4_1-doc.zip
```

The `jar` command is one of the useful Java executables. It became usable when you ran the script and added `C:\j2sdk1.4.1_02\bin` to your path. `jar` stands for *Java Archive*. It is like `Stuffit-Expander`, but you run it from the command line. Fortunately, the `jar` file format is compatible with the `.zip` format, so you can use `jar` to extract any `.zip` archive.

Creating Program Files

Congratulations! You are ready to go. Right now you can run any animated illustration in the book. And please do... they are an essential part of your [Ground-Up Java](#) experience.

The other essential part of your experience is writing your own Java programs. You will do this when you work on the "write-a-program" questions at the end of the chapters, and of course you can write programs that implement your own ideas. We have already recommended that you do this in a directory called `\MyJavaCode`, and have your script "cd" in that directory. But now the question is, how do you create Java code? You will see in [Chapter 2](#) that writing a Java program means creating one or more files, called source files, in plain text format, with names that end with `.java`.

There are two ways to create Java source files:

- Use a general-purpose editor.
- Use an Integrated Development Environment.

You have several general-purpose editors installed on your system, including Notepad and Wordpad. It's a good idea to start with one of these. They are good enough for small programs. Use a fixed-width font like Courier to make your code line up nicely.

As you will learn from experience, you don't just write a program. The development process is an ongoing cycle of writing, testing, and modifying. So when you think you have finished writing your program, don't close your editing window. Leave it around, because in all likelihood you will want to make modifications or fix bugs. (Yes, this could even happen to you.)

After a while, you might get a vague sense that life could be better somehow. You might be ready for an IDE, or Integrated Development Environment. IDEs are products that help you create, maintain, debug, and keep track of Java programs. Many common operations (such as compiling) are achieved with a single button click. There are lots of IDEs on the market, ranging in price from free to expensive. It would be inappropriate to recommend one here, but if you type "Java + IDE" into your favorite Web search engine, you will get plenty of information.

A good IDE is a good thing, and a great IDE will greatly enhance your productivity. But a word of warning: Your goal right now is *not* to create large Java programs efficiently. Your goal is to learn as much as possible about Java. IDEs shield you from repetitive tasks. Before you use them, it's a good idea to spend some time learning all the ins and outs and details of Java, so that you'll know what the IDE is shielding you *from*. Spend some time with a general-purpose editor before you move on to an IDE.

Appendix B: Solutions to the Exercises

Chapter 1

Exercise 1 A cluster of eight bytes can take on approximately 20 quintillion different values. (One quintillion is a 1 followed by 18 zeroes, or 10 to the 18th power.) Estimate the number of different values that a cluster of 16 bytes can have. Just estimate, do not count. Can you think of anything that comes in such quantities?

Solution 1 The exact number of values is 2 to the power of the number of bits. This is 2^{128} , or about 3.4×10^{38} . We can make a good estimate by just squaring the number of possibilities for eight bytes, which is given as approximately 20×10^{18} . The square of that is approximately 400×10^{36} , or approximately 4×10^{38} .

To put this in perspective, there are about 2×10^{11} stars in a typical galaxy, and there are about 10^{10} galaxies in the universe. So 16 bytes can easily store the number of stars in the universe (2×10^{21}).

Exercise 2 The SimCom animated illustration is written in Java. When you run the program, how many virtual machines are at work?

Solution 2 SimCom is a virtual machine that runs on the Java Virtual Machine that runs on your physical computer. So there are two virtual machines.

Exercise 3 Write a SimCom program that adds 255 to the value in byte 31 and stores the result in byte 30. Observe the program's behavior. What do you notice?

Solution 3 The following program adds 255 to the contents of byte 31, and stores the result in byte 30. The program appears in the solutions on the CD-ROM, in answers/ Ch1/Add255.simcom. In addition to the following code, the program also stores the number 1 in byte 29:

```
LOAD 31
ADD 29
STORE 30
HALT
```

SimCom acts as if adding 255 were the same as subtracting 1. We will look at this in more detail in the [next chapter](#).

Exercise 4 Write a SimCom program that computes the square of the value in byte 31 and stores the result in byte 30. What happens when you try to compute the square of 254?

Solution 4 The following program squares the contents of byte 31, and stores the result in byte 30. The program appears in the solutions on the CD-ROM, in answers/Ch1/ Square.simcom:

```
LOAD 31
STORE 29
LOAD 31
ADD 30
STORE 30
LOAD 29
SUB 28
STORE 29
JUMPZ 10
JUMP 2
HALT
```

This program is almost the same as the Times5 program that you saw earlier in [Chapter 1](#). The difference is that instead of using a hard-coded value as the loop counter, the first two lines of this program store the value to be squared in the loop counter.

This program produces 4 as the square of 254.

Exercise 5 What features could be added to SimCom to make it more useful?

Solution 5 This is a subjective issue. It would be reasonable to want more opcodes, especially for multiplying and dividing. More memory would also be good. But bear in mind that, since SimCom forces you to be aware of all features of the architecture, adding more features would just give you more to juggle. The benefit of a high-level programming language such as Java is that you can take advantage of a computer's features without having to think on too low a level.

Chapter 2

Exercise 1 According to [Table 2.1](#), the maximum values for the byte and short data types are 127 and 32767, respectively. Use the Twos-Complement Lab animated illustration to verify this. Which byte and short bit patterns produce the maximum values? In general, which bit pattern produces the maximum value for a two's complement number of N bits?

Solution 1 The maximum-value byte is 01111111. The maximum-value short is 0111111111111111. The general formula is a leading 0 followed by all 1s.

Exercise 2 According to [Table 2.1](#), the minimum values for the byte and short data types are -128 and -32768, respectively. Use the Twos-Complement Lab animated illustration to verify this. What byte and short bit patterns produce the minimum values? In general, what bit pattern produces the minimum value for a two's complement number of N bits?

Solution 2 The minimum-value byte is 10000000. The minimum-value short is 1000000000000000. The general formula is a leading 1 followed by all 0s.

Exercise 3 Launch the Twos-Complement Lab animated illustration by typing `java TwosCompLab`, set the data type to int, and set all the bits to 1. Then set the three bits on the right to 0. Compute the value. Do the same for the byte and short data types. What do you observe?

Solution 3 In each case, the result is -8.

Exercise 4 Launch the Floating-Point Lab animated illustration by typing `java floating.FloatFrame`. Set the rightmost bit to 1 and all other bits to 0. The value represented is 1.4E-45. Try changing various bits' values by clicking on them. Can you create a value that is smaller than 1.4E-45 but still greater than 0?

Solution 4 1.4E-45 is the smallest possible greater-than-zero float value. [Table 2.2](#) says so. Changing any bits in the exponent part yields a bigger power of 2. Changing any bits in the fraction part yields a bigger fraction, unless you set all the fraction bits to 0, which represents an overall value of 0.

Exercise 5 Write a Java application that declares and assigns values to three int variables named x, y, and z. Print out all three values, separated by commas, on a single line.

Solution 5 The following application prints out the values, separated by commas:

```
public class Ch2Q5
{
    public static void main(String[] args)
    {
        int x, y, z;
        x = 10;
        y = 20;
        z = 30;
        System.out.println(x + "," + y + "," + z);
    }
}
```

Exercise 6 *White space* means spaces, tabs, and line-break characters. Type in the VerySimple application from [Chapter 2](#) (reproduced below) and experiment with inserting white space. Does anything change during compilation or execution if you insert extra spaces between `public` and `class`? What if you insert a line break between `public` and `class`? Can you find any adjacent words or symbols such that inserting white space between them changes compilation or execution?

```
public class VerySimple
{
    public static void main(String[] args)
    {
        double age;
        age = 123.456;
    }
}
```

Solution 6 White space between words and symbols has no effect on compilation or execution, unless you put the white space inside a literal string. In that case, of course, the literal string will be changed. This means that you can use white space to make your source code as readable as possible. This is discussed further in [Chapter 3](#), in the section "[White Space and Comments](#)."

Chapter 3

Exercise 1 What happens when a comment appears inside a literal string? (Recall from [Chapter 2](#) that a literal string is a run of text enclosed between double quotes.) What would the following line of code do?

```
System.out.println("A /* Did this print? */ Z");
```

Write a program that includes this line. Does the program print the entire literal string, or does it just print "A Z"?

Solution 1 The program prints the entire literal string. A // or /* inside a literal string doesn't signal the start of a comment.

Exercise 2 What is the value of ~100? What is the value of ~~100? First try to figure it out, and then write a program to print out the values. (Hint: You can figure it out without using pen and paper if you remember something that was discussed in [Chapter 2](#).)

Solution 2 ~100 is -101. ~~100 is 99. Recall from [Chapter 2](#) that to generate the negative of an integer type, invert all its bits and then add one. In other words, if you use ~ to invert an integer's bits, you have almost generated its negative. Almost, but not quite: You still have to add 1. So ~ generates the negative of its argument, minus 1.

The following program performs the computations:

```
public class TildeTest
{
    public static void main(String[] args)
    {
        int n = 100;
        int nTilde = ~n;
        System.out.println("~100 = " + nTilde);
        n = -100;
        nTilde = ~n;
        System.out.println("~-100 = " + nTilde);
    }
}
```

Exercise 3 Write a program that prints out the following values:

```
32 << 3
32 >> 3
32 >>> 3
-32 << 3
-32 >> 3
-32 >>> 3
```

Solution 3 The following program performs the required operations:

```
public class Shift32By3
{
    public static void main(String[] args)
    {
        int x = 32 << 3;
        System.out.println("32 << 3 = " + x);
        x = 32 >> 3;
        System.out.println("32 >> 3 = " + x);
        x = 32 >>> 3;
        System.out.println("32 >>> 3 = " + x);
        x = -32 << 3;
        System.out.println("-32 << 3 = " + x);
        x = -32 >> 3;
        System.out.println("-32 >> 3 = " + x);
        x = -32 >>> 3;
        System.out.println("-32 >>> 3 = " + x);
    }
}
```

The output is

```
32 << 3 = 256
32 >> 3 = 4
32 >>> 3 = 4
-32 << 3 = -256
-32 >> 3 = -4
-32 >>> 3 = 536870908
```

Exercise 4 What are the values of the following expressions? First do the computations mentally. Then write a program to verify your answer.

```
false & ((true^(true&(false|(true|false))))^true)
true | (true^false^false^true&(false|(true&true)))
```

Solution 4 The first expression has the form "false & anything", so its value is false. The second expression has the form "true | anything", so its value is true. The following program verifies this:

```
public class AndAnythingOrAnything
{

```

```
public static void main(String[] args)
{
    boolean a = false & ((true^(true&(false!(true|false))))^true);
    boolean b = true | (true^false^false^true&(false!(true&true)));
    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
}
```

Exercise 5 The following expression looks innocent:

```
boolean b = (x == 0) | (10/x > 3);
```

You can assume x is an int. Write a program that prints out the value of this expression for the following values of x : 5, 2, 0. What goes wrong? (You will see a failure message that you might not be familiar with, because we have not introduced it yet. Don't worry—just try to understand the general concept.) How can you make the code more robust by adding a single character to the expression?

Solution 5 The following program does what the question requires:

```
1. public class Chap3Q5
2. {
3.     public static void main(String[] args)
4.     {
5.         int x = 5;
6.         boolean b = (x == 0) | (10/x > 3);
7.         System.out.println("x=" + x + ", b=" + b);
8.         x = 2;
9.         b = (x == 0) | (10/x > 3);
10.        System.out.println("x=" + x + ", b=" + b);
11.        x = 0;
12.        b = (x == 0) | (10/x > 3);
13.        System.out.println("x=" + x + ", b=" + b);
14.    }
15. }
```

The output from line 7 is "x=5, b=false". The output from line 10 is "x=2, b=true". You don't get any output from line 13. Instead, the JVM returns an error message. Your message may vary based on your JVM rev, but probably you saw the following:
java.lang.ArithmeticException: / by zero at Chap3Qs.main(Chap3Qs.java:12)

When a program prints out a message like this that includes the word "Exception", you know that something has gone wrong. Exceptions are Java's mechanism for indicating program trouble or failure. They're covered in [Chapter 11](#), "Exceptions." The stuff in parentheses at the end of the message says that something went wrong at line 12, so execution was abandoned at that point. The message and a glance at line 12 tell us that we have tried to divide 10 by zero. This is an illegal operation, because dividing by zero is undefined.

To fix the program, just change `|` to `||`. At line 12, the " $x == 0$ " comparison will evaluate to true, so the short-circuit operator will skip the illegal remainder of the expression.

The "Short-Circuit Operators" section of [Chapter 3](#) explained that short-circuit operators let you avoid unnecessary execution of time-consuming code. This question shows that you can also use them to avoid unnecessary execution of code that would generate an error.

Exercise 6 The 32-bit float type is wider than the 64-bit long type. How can a 32-bit type be wider than a 64-bit type?

Solution 6 Longs (64 bits) use two's-complement data representation, and floats (32 bits) use floating-point representation. No matter what data representation is used, there are exactly 2^n possible combinations of n bits. The way to think about this problem is to consider the way that represented numbers are distributed. The long type represents 2^{64} values, evenly distributed along the number line. In other words, the distance between any two consecutive numbers represented by a long is exactly 1.

With floats, the 2^{32} values are not evenly distributed. If you draw a dot on the number line for every number represented by a float, you see a dense cluster near zero. The farther you get from zero, the more sparsely the dots appear. Far out near the extreme positive and negative ends of the range, the dots are very rare indeed. To quantify, the smallest-magnitude float that is greater than zero—in other words, the first number to the right of zero on the number line—is 1.4×10^{-45} . However, the difference between the largest float and the next-smallest float is about 2×10^{31} —a truly astronomical number.

So the 32-bit float type achieves a wider range than the 64-bit long type by distributing its represented values more sparsely.

Exercise 7 Write a program that contains the following two lines:

```
byte b = 6;
byte b1 = -b;
```

What happens when you try to compile the program?

Solution 7 The first line (`byte b = 6;`) is legal. The second line (`byte b1 = -b;`) is a problem. The result of the unary `-` operation is of type `int`, and the code tries to assign an `int` to a `byte`. The compilation will fail. The compiler error may vary depending on your compiler rev, but probably you will get a message that says this:

```
...possible loss of precision: int, required: byte...
```

Chapter 4

Exercise 1 Which of the following are legal method names?

- a. \$25
- b. 25\$
- c. abc_
- d. _ABc

Solution 1 A, C, and D are legal. B is illegal because a method name may not begin with a digit.

Exercise 2 Suppose you want to write a method that returns the diameter of a planet, in millimeters. Since it's your program, you can choose any name you like for the method. Rank the following method names, from worst to best. Use your own judgment as to what makes one method name better or worse than another.

- a. getPlanetDiameter
- b. getSize
- c. getPlanetDiameterMm
- d. getIt
- e. getPlanetSize

Solution 2 Good and bad are subjective. However, a method name that eliminates confusion must be considered better than one that creates confusion, or only eliminates a little confusion. Here are the method names, ranked by order of how much information each name conveys:

```
getIt  
getSize  
getPlanetSize  
getPlanetDiameter  
getPlanetDiameterMm
```

`getIt` tells you nothing at all about what the method does. It's unfortunate how many programmers use similar names and create code that is difficult to understand and expand. The other names tell increasingly more about the method's return value. "Diameter" is better than "Size", because `Size` might be diameter or radius or mass. "DiameterMM" tells us not only the quantity but the units.

Of course, there is a limit to how much a method name should say. The goal is not to maximize the information in the name. The goal is to maximize the *usefulness* of the name. A name that is too long to read easily, or hard to distinguish from a similar name, does not contribute. For example, `getPlanetDiameterMMAsMeasuredByHubbleOnApril112003` is too informative, and it's hard to distinguish from `getPlanetDiameterMMAsMeasuredByHubbleOnApril112003`.

Exercise 3 Suppose a method has the following declaration:

```
static int abc(int x, short y)
```

Suppose this method is called as follows:

```
abc(first, second)
```

Which of the following are legal types for the variables `first` and `second`?

- a. int first, int second
- b. short first, short second
- c. byte first, char second
- d. char first, byte second

Solution 3 B and D are legal. A passed argument may be of any type, provided it is the same as, or narrower than, the type declared by the method. The first argument is declared by the method to be an int, so you can pass a byte, short, char, or int. The second argument is declared by the method to be a short, so you can pass a byte or a short.

Exercise 4 Consider the following method declaration:

```
xyz(double d)
```

Which argument types can a caller pass into this method?

Solution 4 A caller can pass any type that is the same as, or shorter than, the declared type. Since the declared type is double, the caller can pass a byte, short, char, int, long, float, or double.

Exercise 5 In [Chapter 4](#), you learned that if method `iAmVoid` is void, you can't say `int z = iAmVoid();` because there is no value to assign to `z`. What happens if you try? Write a program that does this experiment.

Solution 5 The following application tries to assign a void call to an int:

```
public class AssignVoid  
{  
    static void iAmVoid()  
    {  
    }
```

```
        System.out.println("Hello");
    }

    public static void main(String[] args)
    {
        int z = iAmVoid();
    }
}
```

The compilation fails with the following message: "Incompatible types; found, void, required:int at line 10..." (Your actual message may vary, depending on where your compiler came from.)

Exercise 6 In [Chapter 4](#), you saw the following method:

```
static void print3x(int x)
{
    x = 3*x;
    System.out.println("3 times x = " + x);
}
```

The following code prints out "Now z is 10", not "Now z is 30", because the method modifies its own private copy of the argument:

```
int z = 10;
print3x(z);
System.out.println("Now z is " + z);
```

Write a program that proves this.

Solution 6 The following code proves that the method modifies its own copy, leaving the caller's copy alone:

```
public class ProveCallByValue
{
    static void print3x(int x)
    {
        x = 3*x;
        System.out.println("3 times x = " + x);
    }

    public static void main(String[] args)
    {
        int z = 10;
        print3x(z);
        System.out.println("Now z is " + z);
    }
}
```

Chapter 5

Exercise 1 Rewrite the following code to maximize readability:

```
switch (x)
{
    case 100:
        System.out.println("x is big");
        break;
    case 101:
        System.out.println("x is big");
        break;
    case 10:
        System.out.println("x is medium");
        break;
    case -1000:
        System.out.println("x is negative");
        break;
}
```

Solution 1 The 100 and 101 cases can be combined, and the cases can be arranged in ascending numerical order, to produce the following:

```
switch (x)
{
    case -1000:
        System.out.println("x is negative");
        break;
    case 10:
        System.out.println("x is medium");
        break;
    case 100:
    case 101:
        System.out.println("x is big");
        break;
}
```

Exercise 2 Rewrite the following code to make it cleaner:

```
boolean flag = false;
switch (a)
{
    case 1:
        x = 1000;
        flag = true;
        break;
    case 30:
        y = 1000;
        flag = true;
        break;
}
if (!flag)
    z = 1000;
```

Solution 2 The flag just indicates that the switch has a case that matches its argument. So the "z = 1000" assignment happens only if there was no case to match the switch argument. We can eliminate the flag and move the "z = 1000" assignment into the switch's default case:

```
switch (a)
{
    case 1:
        x = 1000;
        flag = true;
        break;
    case 30:
        y = 1000;
        flag = true;
        break;
    default:
        z = 1000;
        break;
}
```

Exercise 3 What happens when the following code is executed with val equal to 10? 100? 1,000? First, decide just by looking at the source code. Then write a program to verify your answer.

```
switch (val)
{
    case 10:
        System.out.println("ten");
    case 100:
        System.out.println("hundred");
    default:
        System.out.println("thousand");
}
```

Solution 3 The code doesn't have any break statements, so every case will fall through to the next one. The output for 10 is

```
ten
hundred
thousand
```

The output for 100 is

```
hundred
thousand
```

And the output for 1000, which is handled by the default code, is

```
thousand
```

The following program verifies the results. Note the for loop, with multiple action in the update:

```
public class SwitchTest
{
    public static void main(String[] args)
    {
        int val = 10;
        for (int i=0; i<3; i++, val*=10)
        {
            System.out.println("\nTesting " + val + " ... ");
            switch (val)
            {
                case 10:
                    System.out.println("ten");
                case 100:
                    System.out.println("hundred");
                default:
                    System.out.println("thousand");
            }
        }
    }
}
```

Exercise 4 Run the WhileLab animated illustration by typing `java loops.WhileLab`. Try changing the value in the condition in the third line. What do you notice about the final value of `a`?

Solution 4 The final values of `a` are always square numbers.

Exercise 5 The description of WhileLab suggests three exercises, which are repeated here. For each desired result, configure the inputs of WhileLab to produce that result. Then verify your work (and make sure WhileLab is trustworthy) by writing an application that duplicates each while loop. The loops should generate the following results:

- The sum of the numbers 1 through 500, inclusive.
- The sum of the even numbers from 50 through 60, inclusive.
- The product of the first 5 odd numbers.

Solution 5 The sum of 1 through 500:

```
int a = 0;
int b = 1;
while (b <= 500)
{
    a = a+b;
    b = b+1;
}
```

The sum of the even numbers from 50 through 60, inclusive:

```
int a = 0;
int b = 50;
while (b <= 60)
{
    a = a+b;
    b = b+2;
}
```

The product of the first 5 odd numbers (the n^{th} odd number is $2n+1$):

```
int a = 1;
int b = 0;
while (b < 5)
{
    a = a * (2*b+1);
    b = b+1;
}
```

Exercise 6 There is a number game called Hotpo that can entertain you for a few minutes while you're stuck in traffic, waiting for a movie to start, or having dinner with someone really boring. Hotpo stands for Half Or Triple Plus One, and it works like this: Think of an odd number. Now mentally calculate another number, as follows: If the first number was even, the next number is half the first one; if the first number was odd, the next number is 3 times the first number, plus 1. Now you can forget the first number and apply the Half Or Triple Plus One formula to your current number. Keep going until the value reaches 1. Let's try this with a starting number of 5. The series is $5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$. Write a program that plays Hotpo. First, initialize a variable called `n` to the starting value you're interested in. Then enter a loop that prints out each number in the sequence, along with the current step number. For example, the output for 3 would be

```
Step #1: 10
Step #2: 5
Step #3: 16
Step #4: 8
Step #5: 4
Step #6: 2
Step #7: 1
```

Should the program use a while loop or a for loop?

Solution 6 Hotpo is an extreme example of a situation that ought to use a while loop. Remember that for loops are better when you know beforehand how many passes you will make through the loop's body, and while loops are better when you don't know you're done until you're done. If you've played with various values of n , you may have noticed that there's no way to predict whether a certain starting value will need a lot of steps or only a few steps to reach 1. Hotpo defies mathematical analysis. There seems to be no way to predict how many steps a given starting value will require, which means a while loop is ideal. Here is a solution:

```
public class HotpoWhile
{
    public static void main(String[] args)
    {
        int n = 3;
        int nSteps = 0;
        while (n != 1)
        {
            n = (n%2 == 0) ? n/2 : 3*n+1;
            nSteps++;
            System.out.println("Step #" + nSteps + ": " + n);
        }
    }
}
```

Note: If you're ever really bored, try 31.

Exercise 7 What is the value of n after the following code is executed?

```
int n = 1;
outer: for (int i=2; i<10; i++)
{
    for (int j=1; j<i; j++)
    {
        n *= j;
        if (i*j == 10)
            break outer;
    }
}
```

Solution 7 The answer is 24, but the real question is: How did you arrive at the answer? The code is only 10 lines long, and four of those lines are just curly brackets, but the nested loops are intricate enough that working out the answer mentally or on paper is unreliable. It doesn't take long to just type in the code, let it run, and see what happens.

Chapter 6

Exercise 1 The following two declarations are equivalent as far as the compiler is concerned, but one is considered more readable than the other. Which is more readable, and why?

- a. `double dubs[];`
- b. `double[] dubs;`

Solution 1 Format B (`double[] dubs`) is more readable than Format A (`double dubs[]`) because in B, as with all other declarations, the data type (`double[]`) comes first, followed by the variable name (`dubs`). Format A begins with some, but not all, of the data type (`double`). Then comes the variable name, followed by the remainder of the data type (`[]`). So Format A is less readable for two reasons: It does not follow the convention of data type followed by variable name, and it splits the data type into two parts.

Exercise 2 Write a line of code that declares an array of 5 ints and initializes the array to contain the first 5 prime numbers. The code should be a single statement.

Solution 2 `int[] first5Primes = {2, 3, 5, 7, 11};`

Exercise 3 Write a method whose single argument is an array of double. The method should return the average (mean) of the array's components. Write an application that tests the method by passing it an array containing any values you like.

Solution 3 The following code is one possible solution:

```
public class MeanOfArray
{
    public static void main(String[] args)
    {
        double[] theArray = {1.2, 1.3, 1.4, 1.5, 1.6};
        double average = computeAverage(theArray);
        System.out.println("mean = " + average);
    }

    static double computeAverage(double[] doubles)
    {
        double sum = 0;
        for (int i=0; i<doubles.length; i++)
            sum += doubles[i];
        return sum/doubles.length;
    }
}
```

Exercise 4 Write a program that uses the array-averaging method of Question 3. The program should compute and print out the average of an array (you can choose the component values). Then the program should add 100 to each component, and again compute and print out the average.

Solution 4 The following code is one possible solution. The `main` method is long enough that comments are in order:

```
public class Question4
{
    public static void main(String[] args)
    {
        // Create the array.
        double[] theArray = {1.2, 1.3, 1.4, 1.5, 1.6};

        // Compute and print out average.
        double average = computeAverage(theArray);
        System.out.println("mean = " + average);

        // Add 100 to each component.
        for (int i=0; i<theArray.length; i++)
            theArray[i] += 100;

        // Compute and print out new average.
        average = computeAverage(theArray);
        System.out.println("mean = " + average);
    }

    static double computeAverage(double[] doubles)
    {
        double sum = 0;
        for (int i=0; i<doubles.length; i++)
            sum += doubles[i];
        return sum/doubles.length;
    }
}
```

Exercise 5 Write a program that contains a method that creates and returns an array of int containing the first `n` square numbers, where `n` is the method's argument. Test your method by calling it with `n=10`. Your program should print out the index and value of each component, in *descending* order.

Solution 5 The following code is one possible answer. Note that the loop in `main` decrements its loop counter down to and including 0, because of the requirement that the squares should be printed out in descending order:


```
public class DescendingSquares
{
    public static void main(String[] args)
    {
        int[] squares = createArrayOfSquares(10);
        for (int i=squares.length-1; i>=0; i--)
            System.out.println(squares[i]);
    }

    static int[] createArrayOfSquares(int nSquares)
    {
        int[] squares = new int[nSquares];
        for (int i=0; i<nSquares; i++)
            squares[i] = i*i;
        return squares;
    }
}
```

Exercise 6 Write a method that creates a multiplication table. The method should return a two-dimensional array of N by N ints, where N is specified by the method's argument. In the array, the component at [row][col] should have a value of row*col.

Solution 6 The following method creates a multiplication table:

```
static int[][] makeTable(int n)
{
    int[][] table = new int[n][n];
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            table[i][j] = i*j;
    return table;
}
```

Chapter 7

Exercise 1 Name four traits that arrays and objects have in common.

Solution 1 Any four of the following are acceptable answers:

- They contain clusters of data.
- They are created by invoking the keyword `new`.
- They inhabit inaccessible memory.
- They are manipulated indirectly, via references.
- They cannot be passed as array method arguments, but references to them can.
- They are not destroyed explicitly. They are garbage-collected when they have no more references.

Exercise 2 Name two differences between arrays and objects.

Solution 2 Any two of the following are acceptable answers:

- They can contain data of different types.
- They can contain methods as well as data.
- They are related to classes.

Exercise 3 Objects are not passed as method arguments, but references to objects can be passed. When a reference is passed into a method, any changes made to the referenced object by the method should be visible to the method's caller. Write an application to demonstrate this.

Your application will have two classes: `Cat` and `Ager`. The `Cat` class should have a single variable: an `int` called `age`. The `Ager` class should have a method whose signature is `makeOlder(Cat kitty, int nYears)`. This method should add `nYears` to the age of the `Cat` object referenced by `kitty`. Your `main` method should go in the `Ager` class. It should create one instance of each class, set the cat's age, and then use the `Ager`'s method to change the age. Your `main` should then print out the cat's new age, and verify that it really changed.

Solution 3 In file `Cat.java`:

```
public class Cat
{
    int age;
}
```

In file `Ager.java`:

```
public class Ager
{
    void makeOlder(Cat kitty, int nYears)
    {
        kitty.age += nYears;
    }

    public static void main(String[] args)
    {
        Ager myAger = new Ager();
        Cat myCat = new Cat();
        myCat.age = 5;
        System.out.println("Age was " +
            myCat.age);
        myAger.makeOlder(myCat, 2);
        System.out.println("Age became " +
            myCat.age);
    }
}
```

Exercise 4 What happens if you move the `main` method of the previous question from the `Ager` class to the `Cat` class?

Solution 4 You have to run the program by typing "`java Cat`" instead of "`java Ager`". Otherwise the output is the same. The point of this question is that in a multiple-class application, you have to decide where to put your `main` method.

Exercise 5 Write an application that causes a "null pointer exception" failure.

Solution 5 The following application causes a "null pointer exception" failure:

```
public class IFail
{
    int x;

    public static void main(String[] args)
    {
        IFail ref = null;
        ref.x = 10;
    }
}
```

This is just one of an infinite number of possible examples. When you write long programs, "null pointer exception" failures are unavoidable in the course of developing, debugging, and refining your code.

Exercise 6 What does the following application print out?

```
public class Question
{
    static long x;

    public static void main(String[] args)
    {
        Question q1 = new Question();
        Question q2 = new Question();
        q1.x = 10;
        q2.x = q1.x + 20;
        System.out.println("q1.x = " + q1.x);
    }
}
```

Solution 6 The code prints out "q1.x = 30". Since x is static, "q1.x" and "q2.x" are both names for the same variable. The main method could be rewritten as

```
public static void main(String[] args)
{
    Question q1 = new Question();
    Question q2 = new Question();
    Question.x = 10;
    Question.x = Question.x + 20;
    System.out.println("q1.x = " + q1.x);
}
```

This question points out that referring to a static variable via the class name is clearer than referring to it via a reference to an instance of the class.

Chapter 8

Exercise 1 Which of the following hierarchies illustrate a good understanding of the difference between classes and objects? Which ones represent mistaken understanding? The arrows mean "has subclass", so in option A, Shape → Triangle means "class Shape has subclass Triangle."

- Shape → Triangle → RightTriangle
- GreatLiterature → GreatPoem → DivineComedy
- Planet → Continent
- Person → HeadOfState → Emperor
- Person → HeadOfState → Emperor → AugustusCaesar

Solution 1 A and D are good examples. "RightTriangle" is a category that falls within the broader category of "Triangle", which falls within the even broader category of "Shape". Similarly, "Emperor" is a category that falls within the broader category of "HeadOfState", which falls within the even broader category of "Person".

B starts off well: "GreatPoem" is a category that falls within the broader category of "GreatLiterature". But Dante's *Divine Comedy* is not a category. It is an instance of a category. In software, `divineComedy` should be an instance of class `GreatPoem`, which would be a subclass of `GreatLiterature`.

C isn't even close. Certainly, planets contain continents, and both planets and continents are categories of things, but a continent is not a more specific kind of planet. It would not be appropriate for class `Continent` to extend class `Planet`. (It might be appropriate for the two classes to exist, but be unrelated in terms of inheritance. In this case, perhaps `Continent` would have an array of `Planet`.)

E is like B. The last item is an instance of a category, not a category. Emperor is a category, but there was only one Augustus Caesar. So AugustusCaesar could be an instance of class `Emperor`, which extends class `HeadOfState`, which extends class `Person`.

Exercise 2 Which of the following classes have a no-args constructor?

- A)

```
class A { }
```
- B)

```
class B
{
    B() { }
```
- C)

```
class C
{
    C(int x) { }
```
- D)

```
class D
{
    D(int y) { }
    D() { }
```

Solution 2 There are two ways for a class to get a no-args constructor:

- It can define one explicitly.
- It can define no constructors at all. In that case, the compiler provides a default no-args constructor.

A has no constructors, so it is given a default no-args constructor. B and D define their own no-args constructors. C defines a constructor that takes arguments, so it has no no-args constructor.

Exercise 3 Write the code for two classes. The first, called `WaterBird`, has a float variable called `weight`. The class has a single constructor that looks like this:

```
WaterBird(float w)
{
    weight = w;
}
```

Compile this class. Now create the second class, called `Duck`, which extends `WaterBird`. `Duck` has no variables or methods, so it shouldn't take you long to write it. Will `Duck` compile? First, think about the issues involved. Then try to compile `Duck` and see if you were right.

Solution 3 The `Duck` class looks like this:

```
public class Duck extends WaterBird { }
```

It looks innocent enough, but if you've watched enough cartoons, you know that innocent-looking ducks are not to be trusted. This class defines no constructors, so it gets a default no-args constructor that does almost nothing. The constructor doesn't initialize anything (since it contains no code), but it does participate in the chain of construction. Thus, it tries to call the superclass's default constructor, and there we get into trouble.

The `WaterBird` superclass defines a constructor that takes an argument. There is no explicit no-args constructor, and there is no automatic default constructor. So an invisible piece of functionality in an invisible constructor in `Duck` is trying to call something in `WaterBird` that does not exist. When you try to compile `Duck`, you get an error message. The text of the message may vary depending on your compiler, but it will say something like this:

```
Constructor WaterBird() not found in class WaterBird
```

This kind of trouble is called the *constructor trap*. To get out of the trap, add a no-args constructor to the superclass.

Exercise 4 Write some code to demonstrate to yourself the chain of construction. Create an inheritance hierarchy of 4 classes. Give them any names you like. They don't have to have any data or methods, but each one should have a no-args constructor. These constructors should print out a line identifying the current class (something like "Constructing an instance of `WaterBird`"). Your `main()` method should construct a single instance of your lowest-level subclass. What is the output? Does it matter which class contains the `main()` method?

Solution 4 Here is one solution:

```
public class TwoDShape
{
    TwoDShape()
    {
        System.out.println("Constructor for TwoDShape");
    }
}

public class Polygon extends TwoDShape
{
    Polygon()
    {
        System.out.println("Constructor for Polygon");
    }
}

public class Triangle extends Polygon
{
    Triangle()
    {
        System.out.println("Constructor for Triangle");
    }
}

public class RightTriangle extends Triangle
{
    RightTriangle()
    {
        System.out.println("Constructor for RightTriangle");
    }

    public static void main(String[] args)
    {
        new RightTriangle();
    }
}
```

The output is

```
Constructor for TwoDShape
Constructor for Polygon
Constructor for Triangle
Constructor for RightTriangle
```

The application's behavior and output are the same no matter which class owns the `main()` method. However, it seems cleaner to put `main()` in `RightTriangle`. Anyone who reads the code for the first time will see the call to the `RightTriangle` constructor and wonder what class `RightTriangle` looks like. That person's job is easier if the `RightTriangle` class is the one they are already looking at.

In general, in a multiclass application you have some choices as to where to put your `main()` method. As always, think about which choice will be the clearest to someone reading the code for the first time.

Exercise 5 Write some code to demonstrate inheritance polymorphism. Create a superclass class with 3 subclasses. The superclass should have a method that prints out a line identifying the current class (something like "I am a Monster"). Two of the subclasses should override this method to print out a different message (like "I am a Werewolf"). Give the superclass a `main()` method with an array of size 4, typed as the superclass (for example, `Monster[] monsters = new Monster[4];`). Your `main()` should populate the array with references to 4 objects, each with a different class, and then traverse the array, calling your method on each array component. What is the output? Does it matter which class contains the `main()` method?

Solution 5 Here is one solution:

```
public class Monster
{
    void identify()
    {
        System.out.println("I am a monster.");
    }

    public static void main(String[] args)
    {
        Monster[] monsters = new Monster[4];
        monsters[0] = new Monster();
        monsters[1] = new Dragon();
        monsters[2] = new Werewolf();
        monsters[3] = new Cyclops();
        for (int i=0; i<monsters.length; i++)
            monsters[i].identify();
    }
}

public class Dragon extends Monster
{
    void identify()
    {
        System.out.println("I am a dragon.");
    }
}

public class Werewolf extends Monster
{
    void identify()
    {
        System.out.println("I am a werewolf.");
    }
}

public class Cyclops extends Monster
{
    void identify()
    {
        System.out.println("I am a cyclops.");
    }
}
```

The output is

```
I am a monster.
I am a dragon.
I am a werewolf.
I am a cyclops.
```

Again, programmatically it doesn't matter which class gets the `main()` method. As for readability, the major piece of data in `main()` is an array of `Monster`, so it makes sense to put `main()` in `Monster`.

Chapter 9

Exercise 1 Suppose package `superpack` contains subpackage `subpack`. Suppose a source file contains the following line:

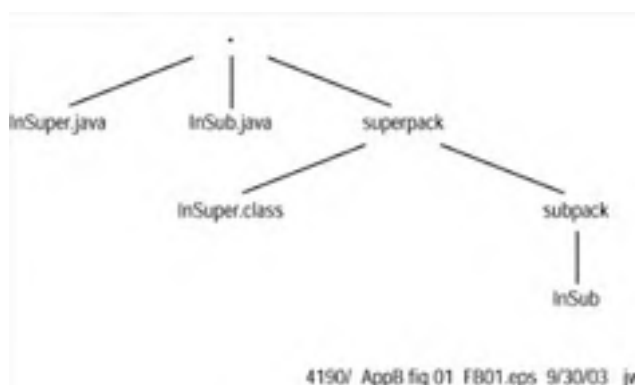
```
import superpack.*;
```

Will this line import classes in `subpack`? Write code to support your answer.

Solution 1 Let's start by creating two classes, one in each package:

```
package superpack;
public class InSuper { }
-----
package superpack.subpack;
public class InSub { }
```

These classes don't do anything, but they are all you need. You can store them in the same directory and compile by typing "`javac -d . *.java`", after which your directory looks like this:



Now you can see what the import line does. Create another class in yet another package:

```
package testpack;
import superpack.*;
public class TestClass
{
    InSub x;
}
```

The class declares a variable of type `InSub`. If the import line doesn't import contents of the subpackage, you should get a compiler error, because the compiler won't know what an `InSub` is. When you compile ("`javac -d . TestClass.java`"), you indeed get an error. This shows that importing "*" does not import subpackages.

Exercise 2 Create a class that illegally tries to read a private variable of another class. What is the point of this exercise?

Solution 2 First let's create and compile the class that owns the private variable:

```
class HasPrivate
{
    private int x;
}
```

Now to try to access `x`:

```
class AccessX
{
    void tryIt()
    {
        HasPrivate hp = new HasPrivate();
        hp.x = 10;
    }
}
```

When you try to compile this class, you get an error message that says something like

```
x has private access in HasPrivate
```

The point of this exercise, and the ones that follow, is to learn to recognize error messages that stem from misuse of the concepts in [Chapter 9](#). This will make it easier to fix bugs when they crop up later on. Meanwhile, you're also getting good practice at thinking in terms of packages and class inheritance structures.

Exercise 3 Create a class that illegally tries to call a default-access method of another class.

Solution 3 Your first class will be called `HasDefMethod`:

```
package aaaaa;
public class HasDefMethod
{
    void deffy()
    {
        System.out.println("deffy here.");
    }
}
```

The method may be called by any class in package aaaaa, so any illegal call attempt will have to come from a different package, like this:

```
package bbbbb;
import aaaaa.HasDefMethod;

public class BadCall
{
    void tryBadCall()
    {
        HasDefMethod h = new HasDefMethod(); // Ok
        h.deffy(); // Won't compile
    }
}
```

The compilation error message says something like this:

deffy() is not public in aaaaa.HasDefMethod; cannot be accessed from outside package

Exercise 4 Create a class that illegally tries to write a protected variable of another class.

Solution 4

You need to create a subclass in a different package from its superclass. First, here's the superclass:

```
package aaaaa;
public class HasProt
{
    protected double d;
} // And here's the subclass:
package bbbbb;
import aaaaa.HasProt;

public class BadWrite extends HasProt
{
    void misuse()
    {
        HasProt other = new HasProt();
        other.d = 3.14159; // Won't compile!
    }
}
```

The compilation error message is

d has protected access in aaaaa.HasProt

Since the subclass is in a different package from the superclass, an instance of the subclass may only access its own copy of a protected variable. In place of the line that doesn't compile, the following would be legal:

```
d = 3.14159;
```

Exercise 5 True or false: If a class has at least one abstract method, the class must be abstract. Write code to support your answer.

Solution 5 True. A class with any abstract methods must be abstract. The following class will not compile:

```
class NotAbstract
{
    abstract void abstractMethod();
}
```

Exercise 6 True or false: If a class is abstract, it must have at least one abstract method. Write code to support your answer.

Solution 6 False. It's okay for an abstract class to have no abstract methods. This isn't stated explicitly in the chapter, but it's easy enough to prove. The following class, which definitely doesn't contain any abstract methods, compiles without error:

```
Abstract class IsAbstract
{
}
}
```

Exercise 7 Write an application that tries to construct an instance of an abstract class. Can you compile the application? Can you execute it?

Solution 7 Here is an abstract class:

```
abstract class Ab { }
```

Here is an attempt to instantiate it:

```
class ConstructAbstract
{
    void constructInstanceOfAbstractClass()
    {

```



```
        Ab theInstance = new Ab();  
    }  
}
```

The compilation error message is something like this:

Ab is abstract; cannot be instantiated

Team LIB

PREVIOUS NEXT

Chapter 10

Exercise 1 Suppose an interface declares three methods. And suppose a class declares that it implements the interface, but in fact it only implements two out of the three methods. What happens when you try to compile the class? (The way to answer this question, of course, is to write an interface and a class.)

Solution 1 You get a compilation error that says your class must be declared abstract. This is a perfectly sensible requirement. The following interface declares three methods:

```
interface Q1Inter
{
    public void a();
    public void b();
    public void c();
}
```

The following class does not completely implement the interface:

```
class Q1Class implements Q1Inter
{
    public void a()
    {
        System.out.println("Method a()");
    }

    public void b()
    {
        System.out.println("Method b()");
    }
}
```

When you compile, you get the following message or something very similar: "Class Q1Class should be declared abstract; it does not define method c() in interface Q1Inter."

Exercise 2 If class A implements an interface, any subclasses of A inherit all the methods specified in the interface. Does this mean that subclasses of A also implement the interface? Write code to discover the answer.

Solution 2 First, let's define the interface:

```
interface Q2Inter
{
    public void x();
}
```

Now here's a superclass that implements the interface:

```
class Q2Superclass implements Q2Inter
{
    public void x()
    {
        System.out.println("Hello from X.");
    }
}
```

And here's a subclass:

```
class Q2Subclass extends Q2Superclass
{
}
```

The subclass does not explicitly declare that it implements the interface, but it inherits an implementation of `x()` from its parent class. Does the compiler believe that `Q2Subclass` implements `Q2Inter`? Let's add some test code somewhere. We need a `main()` method, and we might as well put it in `Q2Subclass`:

```
class Q2Subclass extends Q2Superclass
{
    public static void main(String[] args)
    {
        Q2Subclass subby = new Q2Subclass();
        if (subby instanceof Q2Inter)
            System.out.println("It implements.");
        else
            System.out.println("It does not implement.");
    }
}
```

The application prints out "It implements", indicating that the subclass implicitly implements the interface declared explicitly by the superclass. In other words, interface implementation is a property that is inherited by subclasses.

Exercise 3 Given the following interface:

```
interface InterfaceQ3
{
    void printALine();
}
```

Will the following code compile?

```
class ClassQ3 implements InterfaceQ3
{
    void printALine()
    {
        System.out.println("OK");
    }
}
```

Solution 3 The code will not compile. The sources don't use explicit access modifiers. In the class code, this means `printALine()` has default access. But all methods (and constants) in an interface are public. The error message is something like this:

Method `printALint()` in class `ClassQ3` cannot implement method `printALint()` in interface `InterfaceQ3` w

Exercise 4 Don't worry, the following question requires absolutely no understanding of physics. In fact, it might make you grateful that you chose computer programming instead. Suppose you have the following interface:

```
package physics;
interface PhysicsConstants
{
    public static final double ELECTRON_MASS_KG = 9.11e-31;
    public static final double
        STEFAN_BOLTZMANN_CONSTANT_WATTS_PER_M2 = 5.67e-8;
}
```

What does the following application print out?

```
package physics;

public class Q4 implements PhysicsConstants
{
    public static void main(String[] args)
    {
        System.out.println("The value is " +
            STEFAN_BOLTZMAN_CONSTANT_WATTS_PER_M2);
    }
}
```

Solution 4 Trick question. The code doesn't print out anything, because it does not compile. There are two n's in "Boltzmann", but in the `main()` method there is only one.

The point of this question is to show that human eyes aren't the best mechanism for catching typos in long strings. When you try to compile the application, the compiler immediately finds the typo for you and directs you to the line you need to fix. If you used literal numerical values instead, you would be typing a much shorter string. `"5.67e-8"` only has 7 characters, versus 38 in `"STEFAN_BOLTZMANN_CONSTANT_WATTS_PER_M2"`, so the odds of a typo are 7/38 what they would be if you used the named constant. However, if you type `"5.67e-8"` enough times, you are bound to make a mistake eventually, and the effort of finding the typo would more than cancel out the time you saved by typing the shorter literal numeric value.

Chapter 11

Exercise 1 What happens when you run a program that creates an array of ints and then sets the value of an array component whose index is greater than the length of the array?

Solution 1 The following code tries to set component 60 in an array of length 50:

```
public class Ch11Q1
{
    public static void main(String[] args)
    {
        int[] ints = new int[50];
        ints[60] = 12345;
    }
}
```

The code compiles but throws an exception when executed. The exact message may vary, but the following is typical:

```
java.lang.ArrayIndexOutOfBoundsException at Ch11Q1.main(Ch11Q1.java:6) Exception in thread "main"
```

Exercise 2 What happens when you run a program that creates an array of ints whose length is less than zero?

Solution 2 The following code tries to set create an array of length -25:

```
public class Ch11Q2
{
    public static void main(String[] args)
    {
        int[] ints = new int[-25];
    }
}
```

The code compiles but throws an exception when executed. The exact message may vary, but the following is typical:

```
java.lang.NegativeArraySizeException at Ch11Q2.main(Ch11Q2.java:5) Exception in thread "main"
```

Exercise 3 What happens when you run a program that prints out the result of dividing a non-zero int by zero?

Solution 3 The following code tries to divide 39 by 0:

```
public class Ch11Q3
{
    public static void main(String[] args)
    {
        int and = 39 / 0;
    }
}
```

The code compiles but throws an exception when executed. The exact message may vary, but the following is typical:

```
java.lang.ArithmeticException: / by zero at Ch11Q3.main(Ch11Q3.java:5) Exception in thread "main"
```

Exercise 4 Write a program with a try block that just prints out a message. After the try block, add a catch block that catches `java.io.IOException` (which obviously is not thrown by the try block). Does the code compile? If it compiles, what happens when it runs?

Solution 4 The following code catches an exception type that is never thrown:

```
public class Ch11Q4
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Oye como va");
        }
        catch (java.io.IOException x)
        {
            System.out.println("Caught it.");
        }
    }
}
```

The Java compiler protects you from writing code that can never be run. Since `IOException` is not thrown from the try block, the catch block will never execute. Compilation fails with the following sort of error message:

```
Exception IOException is never thrown in the corresponding try block
```

Exercise 5 Suppose a try block throws many different subclasses of `IOException` (and no other exception types). Suppose you want to catch a few specific subclass types, such as `PrinterIOException` or `ConnectException`. All other exception types should be caught in a safety-net block. Your safety-net block can catch `IOException` or `Exception`. The code will produce the same behavior either way, but the "Catch Blocks and instanceof" section of [Chapter 11](#) says that it's better to use `IOException`. Speculate on why this is true.

Solution 5 Consider a stranger reading your program for the first time. Ask yourself how you can make the source code as easy as possible to understand. This is always a good thing to do, because one day that stranger might be you. (Even if you're reading your own code. It's amazing how you can come back to something you wrote only a few months ago, only to find that you don't remember why you did what you did.)

If your safety-net code catches `IOException`, the stranger will conclude, "The try block throws many kinds of `IOException`." But if your safety-net code catches `Exception`, the stranger will think, "The try block might throw *anything*." So a more specific safety-net catch block gives the stranger more specific information about what the try block might throw.

Exercise 6 What three decisions do you have to make when creating a custom exception subclass?

Solution 6 You have to decide if you want a checked exception or a runtime exception. You have to choose a name for your class. And you have to choose a superclass.

Team LiB

◀ PREVIOUS NEXT ▶

Chapter 12

Exercise 1 In the beginning of [Chapter 12](#), you learned that a good rule of thumb is to use core code when you can and develop original code when you must. Because Java is an object-oriented language, you have a third option, which combines reusing existing code with creating your own. You learned about this option in an earlier chapter. What is it?

Solution 1 The third option is to subclass an existing class. The subclass you create combines preexisting features inherited from the superclass with new features that you implement in the subclass.

Exercise 2 If you write code that calls a deprecated method of one of the core Java classes, what valuable feature of Java can you no longer rely on?

Solution 2 Backward compatibility.

Exercise 3 Suppose you are reading someone else's code and you come across the following lines:

```
Stack myStack = new Stack(); // java.util package
myStack.setSize(100);
```

You decide to look up `setSize()` in the APIs. The comment kindly tells you that class `Stack` is in package `java.util`, so you click on `java.util` in the packages frame, and then you click on `Stack` in the classes frame. You find yourself looking at the class description. You scroll down to the method summaries, and you don't see `setSize` anywhere.

How should you proceed?

Solution 3 There are two ways that class `Stack` can get a `setSize()` method: it can implement it, or it can inherit it. Clearly `Stack` doesn't implement `setSize()`, so it must inherit it.

Scroll down past the end of the method summary section, to the inherited method section. You will see a list of methods inherited from `java.util.Vector`, which is `Stack`'s immediate superclass. There you will see a "setSize" link. Click on it to see the method description on the `java.util.Vector` page.

Alternately, you can scroll up to the top of the `Stack` description page to the inheritance hierarchy. There you will find a link to the `Vector` superclass. Scroll down to the method summaries, where you will find `setSize()`.

Exercise 4 In the section on the `String` class, you learned about the `startsWith(String s)` method, which returns `true` if the executing string object begins with the argument string `s`. It stands to reason that there should be a similar method that tells you whether the executing string object ends with a specified string. Look at the API page for `java.lang.String` and see if such a method exists.

Solution 4 The method does exist. It is called `endsWith()`.

Exercise 5 What happens when you try to compile and execute the following application?

```
public class Ch12Q5
{
    public String toString()
    {
        return "I am an instance of Ch12Q5.";
    }
    public static void main(String[] args)
    {
        Ch12Q5 thing = new Ch12Q5();
        System.out.println(thing);
    }
}
```

Solution 5 The program compiles and executes without error. The call to `System.out.println()` calls `toString()` on `thing`, so the output is

```
I am an instance of Ch12Q6.
```

Exercise 6 What happens when you try to compile and execute the following application?

```
class Ch12Q6
{
    String toString()
    {
        return "I am an instance of Ch12Q6.";
    }
    public static void main(String[] args)
    {
        Ch12Q6 thing = new Ch12Q6();
        System.out.println(thing);
    }
}
```

Solution 6 The difference between this application and the one in Exercise 5 is that the "public" modifiers have been removed from the declarations of the class and the `toString()` method. Now the code won't compile, because `toString()` is public in class `Object`, which is the superclass of `Ch12Q6`. If you override a method, as you have done here with `toString()`, it is illegal to give the subclass version weaker access than the superclass version.

Exercise 7 Look up the explanation of the `equals()` method on the API page of class `java.lang.Object`. The explanation is a bit wordy, but see if you can figure out what it does. (Focus on the last sentence, just before the "Parameters" section.) What is the technical term for what the method does? (Hint: It was introduced in [Chapter 12](#).)

Solution 7 The method checks for reference equality. This isn't so useful, because `equals()` is supposed to check for object equality. No wonder subclasses of `Object` override `equals()`.

Exercise 8 You're not allowed to construct an instance of the `java.lang.Math` class. What happens if you try?

Solution 8 If you write a program that contains the line

```
Math m = new Math();
```

you will get an error message that says something like

```
...constructor Math() has private access in class java.lang.Math.
```

The constructor for the `java.lang.Math` is private. This means that the constructor can be invoked only from within the class itself. This is how the class ensures that you and I can never write code that constructs a `Math` instance.

Exercise 9 The following code models the behavior of a familiar piece of equipment that is used in many games throughout the world. What is the piece of equipment?

```
long rand = 1 + Math.round(Math.random() * 5);
```

Solution 9 The code generates a random int that is ≥ 1 and ≤ 6 , so it simulates shaking dice.

Chapter 13

Exercise 1 In [Chapter 13](#), you learned about the following line:

```
String s = "C:my_backup\temporary\news";
```

What does the following code print out?

```
String s = "C:my_backup\temporary\news";  
System.out.println("***\n" + s + "***");
```

What is the moral of this exercise?

Solution 1 The code prints the following bizarre output:

```
***
```

```
C:my_backup      temporary  
ews
```

The backslash-t is interpreted as a tab, and the backslash-n is interpreted as a newline. The moral is that you always have to use double backslashes in literal strings and chars if you want an actual backslash and not an escape code.

Exercise 2 The code examples in the "[Writing and Reading Data](#)" section defined an int called `i`, a float called `f`, a double called `d`, and so on. But the long was called `n`, which breaks the pattern. You might have expected the long to be called `l`. Why do you think this was not done?

Solution 2 The lowercase letter "ell" looks just like a "one." If you use a lowercase "ell" as a variable name, your code becomes hard to understand. Uppercase "oh" and lowercase "ell" are the two least readable variable names. (Notice how they are spelled out here, in order to make sure there is no confusion. It would be less helpful to tell you that O and I are bad variable names.)

Exercise 3 Write a program that creates a file containing 5,000 random doubles that are ≥ 0 and < 200 .

Solution 3 The following code creates a file that contains 5,000 random numbers in the required range:

```
import java.io.*;  
  
public class Ch13Q3  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            FileOutputStream fos;  
            DataOutputStream dos;  
            fos = new FileOutputStream("RandomDoubles");  
            dos = new DataOutputStream(fos);  
            for (int i=0; i<5000; i++)  
            {  
                double randy = Math.random() * 200;  
                dos.writeDouble(randy);  
            }  
            dos.close();  
            fos.close();  
        }  
        catch (IOException x)  
        {  
            System.out.println("Caught IOException");  
        }  
    }  
}
```

Exercise 4 Write a program that verifies the file you created in the previous exercise. Your program should read the 5,000 doubles, making sure that each falls within the proper range. Your program should also make sure the file contains exactly 5,000 longs.

Solution 4 The following code validates the file that was created in Exercise 3:

```
import java.io.*;  
  
public class Ch13Q4  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            FileInputStream fis =  
                new FileInputStream("RandomDoubles");  
            DataInputStream dis = new DataInputStream(fis);  
            boolean readBad = false;  
            for (int i=0; i<5000; i++)  
            {  
                double randy = dis.readDouble();  
                if (randy < 0 || randy > 200)  
                {  
                    readBad = true;  
                    System.out.println("Read bad double: " +  
                        randy);  
                }  
            }  
        }  
        catch (IOException x)  
        {  
            System.out.println("Caught IOException");  
        }  
    }  
}
```



```
                randy);
            }
        }
        if (!readBad)
            System.out.println("File is valid.");
    }

    catch (IOException x)
    {
        System.out.println("Caught IOException");
    }
}
}
```

The for loop reads 5,000 doubles from the file and checks their range. If a double is out of range, the "Read bad double" message is printed out and the variable `readBad` is set to `true`. After the loop, `readBad` is checked. If it never got set to `true`, the file is considered valid.

Exercise 5 Look up the API documentation for the `java.io.File` class. An instance of this class contains information about an individual file. One of the methods of the class tells you the length in bytes of a file. Use this method to determine the number of bytes in the file you created in Exercise 3.

Solution 5 The following program uses the `File` class to read the size of the file created in Exercise 3:

```
import java.io.*;

public class Ch13Q5
{
    public static void main(String[] args)
    {
        File f = new File("RandomDoubles");
        System.out.println("Length = " + f.length());
    }
}
```

The code prints out "Length = 40000". This is to be expected. The file contains 500 doubles, and each double is 8 bytes.

Chapter 14

Exercise 1 The first code example in [Chapter 14](#) used the following code to set a frame's background color:

```
setBackground(new Color(128, 128, 128));
```

Describe the color that this line creates.

Solution 1 The color has equal levels of red, green, and blue, so it will be some kind of gray. Since the levels are halfway between the minimum (0) and the maximum (255), the gray will be about halfway between black and white: a neutral gray, neither dark nor light.

Exercise 2 Run Color Lab, and adjust the scrollbars so that the displayed color matches something you can see (a piece of clothing you're wearing, or something on your desk, or anything else you like). Now write an application that displays a frame whose interior is the color you've chosen.

Solution 2 The following code shows a frame whose interior matches the color of the shirt I was wearing when I wrote this.

```
import java.awt.*;

public class EmptyFrame extends Frame
{
    EmptyFrame()
    {
        setBackground(new Color(0, 217, 255));
        setSize(300, 300);
    }

    public static void main(String[] args)
    {
        EmptyFrame em = new EmptyFrame();
        em.setVisible(true);
    }
}
```

Remember that in addition to setting the background color, you have to call `setSize()` and `setVisible()`. Otherwise the frame cannot be seen.

Exercise 3 One of the code examples in [Chapter 14](#) used the `getSize()` method, which `Frame` inherits from one of its superclasses. Use the API to find out which superclass implements the method.

Solution 3

```
java.awt.Component
```

Exercise 4 Write a program that draws a five-pointed star. Your frame should be 400 x 400 pixels. The coordinates of the star's points are (200, 375), (97, 58), (366, 254), (34, 254), and (303, 58). The easy way is to write a `paint()` method that calls `drawLine()` five times. But that approach isn't ideal, because you have to type each x and each y twice. (Each point is the end of two lines, so it appears in two `drawLine()` calls.) Typing data, code, or anything else more than once is considered bad style. If one of the copies has a typo and doesn't match the original precisely, your program won't function correctly. To avoid duplication of data, your program should have two `int` arrays, defined as follows:

```
int[] xs = {200, 97, 366, 34, 303};
int[] ys = {375, 58, 254, 254, 58};
```

Your `paint()` method should have a loop that accesses these arrays. `drawLine(...)` should appear only once in your code, inside the loop.

Solution 4 The following program uses a loop to draw a five-pointed star:

```
1. import java.awt.*;
2.
3. public class Supe extends Frame
4. {
5.     int[] xs = {352, 106, 200, 294, 48};
6.     int[] ys = {151, 329, 40, 329, 151};
7.
8.     Supe()
9.     {
10.        setSize(400, 400);
11.    }
12.
13.    public void paint(Graphics g)
14.    {
15.        for (int startPoint=0;
16.            startPoint<xs.length;
17.            startPoint++)
18.        {
19.            int endPoint = startPoint + 1;
20.            if (endPoint == xs.length)
21.                endPoint = 0;
22.            g.drawLine(xs[startPoint], ys[startPoint],
23.                    xs[endPoint], ys[endPoint]);
24.        }
}
```

```
25.     }
26.
27.     public static void main(String[] args)
28.     {
29.         (new Supe()).setVisible(true);
30.     }
31. }
```

Lines 19-21 can be replaced by the following single line:

```
Int endPoint = (startPoint+1) % xs.length;
```

Exercise 5 Write a program that lists all the font families that are available on your computer.

Solution 5 The following program lists all the available font families.

```
import java.awt.GraphicsEnvironment;

public class ListFonts
{
    public static void main(String[] args)
    {
        GraphicsEnvironment grenv =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        String[] names =
            grenv.getAvailableFontFamilyNames();
        for (int i=0; i<names.length; i++)
            System.out.println(names[i]);
    }
}
```

Chapter 15

Exercise 1 Suppose you use the following code to create a checkbox:

```
Checkbox cbox = new Checkbox("Ok", true);
```

What is the checkbox's state after you click on it 20,000 times?

Solution 1 The checkbox's initial state is `true`. After you click on it an even number of times, it is `true` again. There are two ways to get the answer: thinking about it, or doing it. If you chose the second way, you might not be cut out to be a computer programmer.

Exercise 2 In the "Checkboxes" section of [Chapter 15](#), the `Boats` application is 30 lines long. The code isolates literal strings in an array near the top of the listing. You saw how this approach, along with the use of a loop to create the checkboxes, results in more maintainable code. Rewrite the code to eliminate the loop and the string array. In place of the loop in the constructor, just create three checkboxes one by one. How many lines of code does your new application have?

Solution 2 Here is the rewritten code:

```
1. import java.awt.*;
2.
3. class BoatsNoLoop extends Frame
4. {
5.     Checkbox[] cboxes;
6.     Button btn;
7.
8.     BoatsNoLoop()
9.     {
10.         setLayout(new FlowLayout());
11.
12.         cboxes = new Checkbox[3];
13.         cboxes[0] = new Checkbox("a small boat");
14.         add(cboxes[0]);
15.         cboxes[1] = new Checkbox("a medium boat");
16.         add(cboxes[1]);
17.         cboxes[2] = new Checkbox("a large boat");
18.         add(cboxes[2]);
19.         btn = new Button("Add to shopping cart");
20.         add(btn);
21.
22.         setSize(600, 200);
23.     }
24.
25.     public static void main(String[] args)
26.     {
27.         new BoatsNoLoop().setVisible(true);
28.     }
29. }
```

This version is only 29 lines, one line shorter than the original. The point is that a shorter program is not necessarily easier to read or maintain than a longer version. If you still aren't convinced that the original version is better, try Exercise 3.

Exercise 3 This is an extension of Exercise 2. Suppose you need to change the `Boats` application so that instead of offering three sizes (small, medium, and large), it offers ten (rubber duck, sponge, tiny, small, kinda small, medium, kinda large, large, huge, titanic). How does this affect the size of the code as it appears in the "Checkboxes" section of [Chapter 15](#)? How does it affect the size of the code that you wrote for Exercise 2?

Solution 3 The loop-based code will probably become longer by two lines, because the array of literal strings now has 10 members:

```
String[] sizes = {"rubber duck", "sponge", "tiny", "small", "kinda small",
"medium", "kinda large", "large", "huge", "titanic"};
```

The no-loop version grows by two lines for every additional size option (one line to construct a checkbox, another to call `add()`). Seven new options were added, so the code grows by 14 lines, or 50%.

Note Programs usually start out small, and gradually grow as they are required to support more and more functionality. Giving structure to a small program is always worth the effort. The payoff might not be immediately obvious, but it will become more and more evident as time goes by. In Exercise 2, the well-structured program was actually longer than the unstructured version. But when the code needed to support more functionality, the unstructured version grew by 50% while the structured version grew by about 7%.

Exercise 4 Write an application that displays a frame with a menu bar. The bar should have the following menus:

An Edit menu with items Copy and Cut.

A File menu with items Close, Exit, and Open.

A Help menu with item Help. Assume that clicking on this item will display a helpful dialog.

A Whatever menu with items Stuff and Nonsense. The Nonsense item should be a submenu with items Ordinary Nonsense and Extreme Nonsense.

Make sure that your GUI follows the guidelines listed at the end of the "Menus" section.

Solution 4 Here is one solution:

```
import java.awt.*;

class Q4 extends Frame
{
    public Q4()
    {
        MenuBar mbar = new MenuBar();

        Menu fileMenu = new Menu("File");
        fileMenu.add("Open...");
        fileMenu.add("Close");
        fileMenu.add("Exit");
        mbar.add(fileMenu);

        Menu editMenu = new Menu("Edit");
        editMenu.add("Cut");
        editMenu.add("Copy");
        mbar.add(editMenu);

        Menu whateverMenu = new Menu("Whatever");
        whateverMenu.add("Stuff");
        Menu nonsenseMenu = new Menu("Nonsense");
        nonsenseMenu.add("Ordinary Nonsense");
        nonsenseMenu.add("Extreme Nonsense");
        whateverMenu.add(nonsenseMenu);
        mbar.add(whateverMenu);

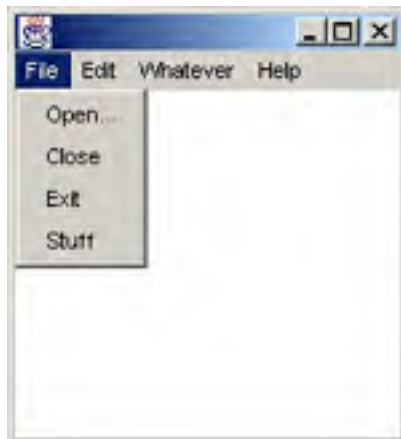
        Menu helpMenu = new Menu("Help");
        helpMenu.add("Help...");
        mbar.add(helpMenu);

        setMenuBar(mbar);

        setSize(300, 200);
    }

    public static void main(String[] args)
    {
        (new Q4()).setVisible(true);
    }
}
```

The following illustrations show the four menus, File, Edit, Whatever, and Help:





Exercise 5 Write a program that creates a GUI that looks like the following illustration. The text in the text area should be set programmatically by a single call to the text area's `append()` method. The call should come directly after the text area is constructed.



Solution 5 Here is one solution:

```
1. import java.awt.*;
2.
3. class Q5 extends Frame
4. {
5.     public Q5()
6.     {
7.         setLayout(new FlowLayout());
8.         TextArea ta = new TextArea(10, 30);
9.         ta.append("Hello\nWorld");
10.        add(ta);
11.
12.        setSize(550, 220);
13.    }
14.
15.    public static void main(String[] args)
16.    {
17.        (new Q5()).setVisible(true);
18.    }
19. }
```

The text is set in line 9. The thing to notice is the newline character, which puts in a line break. It would be wrong to replace line 9 with this:

```
ta.append("Hello");
ta.append("World");
```

Line breaks are inserted only when you explicitly append a newline character. So with the substitution, you would see a single line of text that read "HelloWorld". There wouldn't even be a space between the words.

Exercise 6 Using the API page for `java.awt.FlowLayout`, determine how to create a flow layout manager that right-justifies its cluster of components rather than centering it.

Solution 6 Use the following constructor:

```
new FlowLayout(FlowLayout.RIGHT)
```

Exercise 7 The `java.awt.Component` class, which is a superclass of `java.awt.Button`, has a method called `setSize(int width, int height)`. The method's documentation says that it resizes the component so that its size is `width` times `height`.

What do you expect the following code to do? First, read the listing and decide on your answer. Then, type in the code and run it. Did you see what you expected to see?

```
import java.awt.*;

class Q7 extends Frame
{
    public Q7()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Abcde");
        btn.setSize(500, 500);
        add(btn);
        setSize(700, 700);
    }

    public static void main(String[] args)
    {
        (new Q7()).setVisible(true);
    }
}
```

Solution 7 The code seems to create a large (500 x 500) button in a 700 x 700 frame. But actually, the button's size is perfectly ordinary. The Flow layout manager sets the size of the button to its preferred size.

Exercise 8 This entire chapter has been about components that are installed inside containers. The [previous chapter](#) was about painting. What happens if a frame that contains components also has a `paint()` method that paints a part of the screen that is occupied by a component? Write a program that will reveal the answer.

Solution 8 The frame in the following application has a button, as well as a `paint()` method. The `paint()` method draws diagonal blue lines.

```
import java.awt.*;

class PaintPlusComponent extends Frame
{
    public PaintPlusComponent()
    {
        setLayout(new FlowLayout());
        add(new Button("Apply"));
        setSize(300, 200);
    }

    public void paint(Graphics g)
    {
        g.setColor(Color.blue);
        for (int i=0; i<500; i+=10)
            g.drawLine(i, 0, 0, i);
    }

    public static void main(String[] args)
    {
        (new PaintPlusComponent()).setVisible(true);
    }
}
```

The following illustration shows the GUI. As you can see, the button is superimposed over the painted lines. Whenever a component and a `paint()` method are both responsible for the same part of the screen, the component wins.



Chapter 16

Exercise 1 Write a program that displays a frame. The frame's `paint()` method should draw something simple. The application should also maintain a count of the number of times `paint()` is called. This count should be printed out every time `paint()` is called. Execute your application, and use it to help determine whether `paint()` is called when:

- The application starts up.
- The frame is minimized/iconified.
- The frame is restored to normal size after being minimized/iconified.
- The frame is restored to normal size after being minimized/iconified.
- The frame is moved.
- The frame is partially covered by another frame.
- The frame is uncovered.

Solution 1 The following code prints a message whenever `paint()` is called:

```
import java.awt.*;

public class Ch16Q1 extends Frame
{
    int    nCallsToPaint;

    Ch16Q1()
    {
        setSize(300, 300);
    }

    public void paint(Graphics g)
    {
        g.setColor(Color.cyan);
        g.drawLine(100, 100, 200, 200);
        nCallsToPaint++;
        System.out.println(nCallsToPaint +
                           " calls to paint()");
    }

    public static void main(String[] args)
    {
        (new Ch16Q1()).setVisible(true);
    }
}
```

The `paint()` method is called when the application starts up, and when it is restored after being minimized/iconified. It is also called when the frame is uncovered. It is *not* called when the frame is moved. Depending on your system, it may or may not be called when the frame is covered.

Exercise 2 Every Java thread is represented by an instance of the `java.lang.Thread` class. You can get a reference to the currently running thread by calling the `currentThread()` static method of the `Thread` class. Threads have names. The class has a method called `getName()`, which returns the name as a string. So you can print out the name of the current thread by calling

```
System.out.println(Thread.currentThread().getName());
```

Write a simple frame application that makes this call in its `main()` method and in its `paint()` method. Verify that `main()` and `paint()` are executed in different threads.

Solution 2 The following application prints the name of the current thread in `main()` and `paint()`:

```
import java.awt.*;

public class Ch16Q2 extends Frame
{
    Ch16Q2()
    {
        setSize(300, 300);
    }

    public void paint(Graphics g)
```



```
{
    g.setColor(Color.cyan);
    g.drawLine(100, 100, 200, 200);
    System.out.println("paint() thread is called:");
    System.out.println(Thread.currentThread().getName());
}

public static void main(String[] args)
{
    System.out.println("main() thread is called:");
    System.out.println(Thread.currentThread().getName());
    (new Ch16Q2()).setVisible(true);
}
}
```

Exercise 3 Write an application that adds the same action listener to a button *twice*. For example, if `myButton` is the button and `myListener` is the action listener, your code would contain the following lines:

```
myButton.addActionListener(myListener);
myButton.addActionListener(myListener);
```

Your listener's `actionPerformed()` method should print out a message to tell you that it got called. If you press the button once, do you expect the message to be printed out once or twice? Run your application to see if you guessed right.

Of course, in real life there would never be a good reason for doing this. But you might do it by accident. For example, you might paste the line into your source code twice by accident. So it's good to know in advance what the symptom will be, so that you can recognize it and fix the problem if it ever comes up.

Solution 3 Here is the listener class:

```
import java.awt.event.*;

class Aclis implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("actionPerformed() was called.");
    }
}
```

And here is the application class:

```
import java.awt.*;

public class Ch16Q3 extends Frame
{
    Ch16Q3()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Push Me");
        Aclis ac = new Aclis();
        btn.addActionListener(ac);
        btn.addActionListener(ac);
        add(btn);
        setSize(300, 300);
    }

    public static void main(String[] args)
    {
        (new Ch16Q3()).setVisible(true);
    }
}
```

When you push the button, the message is printed out twice.

Exercise 4 Suppose a class has an `actionPerformed()` method, as specified by the `ActionListener` interface, but the class does not state that it implements the interface. Can an instance of the class be used as a button's action listener?

Solution 4 The following class contains an `actionPerformed()` method, but it does not declare that it implements the `ActionListener` interface:

```
import java.awt.event.*;

class NotAnActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("actionPerformed() was called.");
    }
}
```

Since the class has the right kind of method, you might be tempted to use it as an action listener:

```
. . .
Button btn = new Button("OK");
NotAnActionListener naal = new NotAnActionListener();
btn.addActionListener(naal);
. . .
```

This code will not compile. For a class to be eligible to be an action listener, it is not enough for it to provide an `actionPerformed()` method, because that alone does not mean that it implements the `ActionListener` interface.

Exercise 5 Run Nim Lab by typing `java events.NimLab`. Select Disable Buttons..... and play the game. This version is the result of three rounds of improvements made to the original program. What additional improvements can you suggest? Think about how the game could be modified to make the GUI easier and more natural.

Solution 5 Here are some possible improvements:

- Add a Restart button.
- Provide notification when a player wins.
- Eliminate the buttons. Players would click on a coin to remove it. This would provide direct manipulation of the coins, rather than the indirect manipulation that the buttons provide.

Do you have any other ideas? E-mail them to groundupjava@squares.com, and they might be included in the next revision of this book (with your name mentioned).

Exercise 6 The various event classes (`ActionEvent`, `ItemEvent`, etc.) all inherit the `getSource()` method from a superclass. Use the API pages to determine the name of that superclass.

Solution 6 `java.util.EventObject`.

Exercise 7 Write an application with a GUI that contains a choice and a text area. When the choice is activated, a message should be written to the text area, stating the choice's selected index.

Suggested design: Your frame should contain a panel (at North) that contains the choice. The text area should be at South. If you need a guideline, the `TextAreaNim` program in the "Improving the GUI" section has a similar structure.

Solution 7 Here's the code:

```
import java.awt.*;
import java.awt.event.*;

public class Ch16Q7 extends Frame implements ItemListener
{
    private Choice    choice;
    private TextArea ta;

    Ch16Q7()
    {
        Panel pan = new Panel();
        choice = new Choice();
        choice.add("Dragons");
        choice.add("Centaur's");
        choice.add("Unicorns");
        choice.add("Manticores");
        choice.addItemListener(this);
        pan.add(choice);
        add(pan, "North");

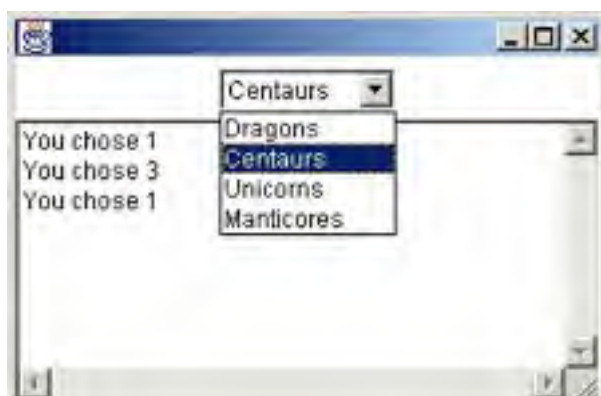
        ta = new TextArea(40, 20);
        add(ta, "Center");

        setSize(300, 200);
    }

    public void itemStateChanged(ItemEvent e)
    {
        ta.append("You chose " +
                choice.getSelectedIndex() +
                "\n");
    }

    public static void main(String[] args)
    {
        (new Ch16Q7()).setVisible(true);
    }
}
```

The following illustration shows the GUI for Exercise 7.



Exercise 8 Write an application with a GUI that contains a text field and a text area. When the user presses the Enter key in the text field, the text field's contents should be copied into text area, followed by a newline character.

Your event-handling code will need to retrieve the contents of the text field. You do that by calling the text field's `getText()` method, which returns a string.

Suggested design: Your frame should contain a panel at North that contains the text field. The text area should go at Center.

Solution 8 Here's the code:

```
import java.awt.*;
import java.awt.event.*;

public class Ch16Q8 extends Frame implements ActionListener
{
    private TextField    tf;
    private TextArea    ta;

    Ch16Q8()
    {
        Panel pan = new Panel();
        tf = new TextField("Type Here", 20);
        tf.addActionListener(this);
        pan.add(tf);
        add(pan, "North");

        ta = new TextArea(40, 20);
        add(ta, "Center");

        setSize(300, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        ta.append(tf.getText() + "\n");
    }

    public static void main(String[] args)
    {
        (new Ch16Q8()).setVisible(true);
    }
}
```

The following illustration shows the GUI for Exercise 8.



Chapter 17

Exercise 1 Write a program that creates a frame with a File menu. The menu should have two items, Save... and Exit. When Save... is selected, the code should display a file dialog box, configured for saving a file. When the user has specified a file via the dialog box, your code should output the name of the file. All the information you need is on the API page for `java.awt.FileDialog`.

Solution 1 Here's the code:

```
import java.awt.*;
import java.awt.event.*;

class SaverFrame extends Frame implements ActionListener
{
    private MenuItem saveMI, exitMI;

    public SaverFrame()
    {
        // Build menu.
        MenuBar mbar = new MenuBar();
        Menu fileMenu = new Menu("File");
        saveMI = new MenuItem("Save...");
        saveMI.addActionListener(this);
        fileMenu.add(saveMI);
        exitMI = new MenuItem("Exit");
        exitMI.addActionListener(this);
        fileMenu.add(exitMI);
        mbar.add(fileMenu);
        setMenuBar(mbar);
        setSize(300, 150);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == exitMI)
            System.exit(0);

        FileDialog dia = new FileDialog(this, "Save Your Work",
                                       FileDialog.SAVE);
        dia.setVisible(true);
        String fileName = dia.getFile();
        if (fileName == null)
            System.out.println("You canceled the dialog.");
        else
            System.out.println("You chose file " + fileName + "
                               in " + dia.getDirectory());
    }

    public static void main(String[] args)
    {
        (new SaverFrame()).setVisible(true);
    }
}
```

The following illustration shows the file dialog, configured for saving.



Exercise 2 The `FileDialog` class has a `setDirectory()` method that controls which directory the dialog box will display. Look up the method description in the API to become familiar with how it works. Modify the final project code so that when the file dialog box appears, it displays one of the directories on your computer where you have stored some of your own Java source code. This will make it easier to display your own work.

Solution 2 Let's say you want the dialog to display the directory `C:\MyCode\Ch7_Exercises`. In `actionPerformed()`, change the code that constructs the file dialog, which in its original form looks like this:

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == openMI)
    {
        if (dialog == null)
            dialog = new FileDialog(this, "Source File",
                                   FileDialog.LOAD);

        dialog.setVisible(true); // Modal
    }
    ...
}
```

Add the `setDirectory()` call immediately after the dialog is constructed, before it is made visible:

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == openMI)
    {
        if (dialog == null)
        {
            dialog = new FileDialog(this, "Source File",
                                   FileDialog.LOAD);
            dialog.setDirectory("C:\\MyCode\\Ch7_Exercises");

            dialog.setVisible(true); // Modal
        }
    }
    ...
}
```

Tip Remember that in Java literal strings, single backslashes are escape characters that have special significance. That's why the argument to the `setDirectory()` call is `C:\\MyCode\\Ch7_Exercises` and not `C:\MyCode\Ch7_Exercises`.

Exercise 3 Write an application that displays a canvas subclass in a frame, at Center. The frame does not contain any other components.

Use the following code as the `paint()` method for the canvas subclass:

```
1. public void paint(Graphics G)
2. {
3.     g.setFont(new Font("Serif", Font.PLAIN, 24));
4.     g.setColor(Color.blue);
5.     g.drawString("Look at this!", 0, 0);
6. }
```

Run the program. Do you see what you expected to see? How do you explain the results?

Now change line 5 to this:

```
g.drawString("A bluejay in a quagmire", 0, 0);
```

Now do you see what you expected to see? Again, how do you explain the results?

Solution 3 Here's the code:

```
import java.awt.*;

class TextCanvas extends Canvas
{
    public void paint(Graphics g)
    {
        g.setFont(new Font("Serif", Font.PLAIN, 24));
        g.setColor(Color.blue);
        g.drawString("A bluejay in a quagmire", 0, 0);
    }

    public static void main(String[] args)
    {
        Frame fr = new Frame();
        TextCanvas tc = new TextCanvas();
        fr.add(tc, "Center");
        fr.setSize(250, 250);
        fr.setVisible(true);
    }
}
```

The y-coordinate argument of the `drawString()` method of class `Graphics` specifies the vertical position of the *baseline* of the text. If the baseline is 0, you will only see those parts of the text that descend below the baseline. In the string "Look at this!", there are no descenders. In "A jay in a quagmire", there is one occurrence of each character that descends: j, y, q, and g. The following illustration shows the GUI with the misplaced baseline. You can see the bottom portions of those letters, hanging down from the top of the canvas.



Exercise 4 The `FancySrcCanvas` class has an array of Java keywords. In that array, `throws` comes before `throw`. Otherwise, the list is alphabetical. Why does `throws` come before `throw`?

Solution 4 The code that looks for keywords checks every position in every source line to see if it begins with a keyword. If it finds a match, it overpaints the keyword in the appropriate color. If `throw` came before `throws`, consider what would happen to the following line:

```
void printPaycheck(Employee emp) throws IOException
```

The code would overpaint `throw`, but the `s` would remain black.

Exercise 5 There are several situations in which the project code would improperly draw text in the keyword color. How many of these situations can you name?

Solution 5 The keyword-finding code ignored line comments—that is, comments beginning with a double slash. But it does nothing about comments that begin with slash-star (`/*`) and end with star-slash (`*/`). Any keyword that appeared in such a comment would be overpainted in the keyword color. The following line would look especially strange:

```
/* Let's go forward despite stiff competition. */
```

The "for" in "forward" and the "if" in "stiff" would appear in the keyword color.

Keywords might coincidentally appear in literal strings, for example:

```
System.out.println("while I was dreaming ...");
```

Lastly, keywords might be embedded in the names of classes, variables, or methods:

```
class Republic extends Country { ... }
```

Exercise 6 How would you modify the project code so that `null`, `true`, and `false` are not rendered in the keyword color?

Solution 6 Delete them from the `keywords` array in `FancySrcCanvas`.

Glossary

A

abstract (keyword)

An abstract method has no body. It may not be instantiated. If a class contains any abstract methods, the class itself must be declared abstract.

access modifiers

Keywords that set the access level of classes, data, and methods.

accessors

A method that supports data hiding. It has an empty argument list and returns a data value. By common convention, the name of an accessor method begins with `get`, followed by the property to be retrieved.

additive primary colors

The primary colors of video screens (red, green, and blue). They combine to form yellow, cyan, and magenta.

allocation

Assigning memory for use as objects or arrays.

analog circuit

Non-digital circuit, where precise voltage values are significant.

application

A Java program that's executed in a Java Virtual Machine, consisting of one or more classes.

array

A cluster of variables (components) that are all of the same type. The array has a name, but its individual components do not.

ASCII

An abbreviation for American Standard Code for Information Interchange. ASCII encodes all the characters in American English, plus punctuation marks, into the range 0-127. The range 128-255 encodes symbols such as accented vowels, which are used in western European languages, as well as some Greek characters, line-drawing symbols, and some others.

assembler

Any program that translates assembly code into base-2 instructions.

assembly language

The language of programming with op-codes. Typically, one line of assembly language code corresponds to a single computer instruction.

B

baseline

The imaginary horizontal line on which the bodies of text characters rest.

binary operators

Numeric or boolean operators that take two operands.

bit

The smallest unit of memory, capable of storing 0 or 1. Abbreviation of "binary digit."

Bitwise operation

Operation in which operands are treated as collections of unrelated individual bits. Only performed on integer data types.

block

A contiguous piece of code that begins with an open curly bracket and ends with a matching closed curly bracket.

boolean (keyword)

A primitive data type that represents `true` or `false`.

bounding box

The smallest rectangle that encloses an oval.

byte (keyword)

An 8-bit signed integer primitive data type.

bytecode

The instruction code for the Java Virtual Machine.

C

catch block

Block of code, following a try block, that handles exceptions of a single type.

chaining

The technique of connecting data streams together.

chain of constructors

The mechanism whereby all constructors begin by invoking a constructor of the superclass.

class files

Bytecode output files produced by the compiler.

class loader

Mechanism that finds class files, reads them, and translates them into internal representations.

classpath

A list of directories that contain package structures.

comments

Text used by programmers to help readers understand the meaning of the code.

compiled language

A programming language that must be translated into computed binary. Unlike assembly language, generally a line of source code does not correspond to a single instruction.

component

A GUI device that presents user input to programs and displays program information to users. Standard GUI components include buttons, text fields, scrollbars, and menus. Also: An array member.

conditional code

Code that's executed only when a boolean criterion is satisfied.

construction

Creation of an object or array.

constructor

Code that creates and initializes an instance of a class.

container

A component that can contain other components.

D

data hiding

The practice of making the data of a class as inaccessible to other classes as possible.

debug code

Code whose purpose is to tell the developer about what is going on inside a program.

declaration

Code that tells the compiler the type of a variable or the return type, argument types, and exception types of a method.

default access

Mode that grants access to all classes in the same package as the class that defines the default feature.

default constructor

A no-args constructor created by the compiler for any class that does not have any constructors.

deprecated method

A method that was introduced in an early revision of Java and should not be used.

destination directory

The directory where the compiler will store a package structure.

dialog box

A window, subordinate to its program's main frame, that is used for brief user interaction.

digital circuit

A circuit where voltages represent 0 or 1.

digital computers

Computers composed of digital circuits.

double (keyword)

64-bit floating-point primitive data type.

E

ellipsis

Three dots (...). In a GUI component, an ellipsis indicates that activating the component will cause the display of a new window or dialog.

empty string

An instance of the String class with zero characters.

event-driven program

A program that acts mainly in response to user input.

events

The mechanism by which components inform listeners that they have been activated.

exception

An object that is thrown to indicate an unusual or error state. The throwing of an exception diverts the normal flow of program control.

F

falling through

In `switch` code, continuing from one case to the next in the absence of a `break` statement.

field

A data variable in a class.

file separator

The character that appears between elements in a full pathname.

final (keyword)

A final class may not be subclassed. A final method may not be overridden. A final variable may not be modified after it is initialized.

flag

A `boolean` variable used to indicate program status.

float (keyword)

A 32-bit floating-point primitive data type.

Team LIB

◀ PREVIOUS

NEXT ▶

G

garbage collection

The automatic recycling of unusable objects.

garbage collection thread

The thread that implements garbage collection.

GUI thread

In applications with GUIs, the thread that paints components and notifies event listeners.

Team LIB

◀ PREVIOUS

NEXT ▶

I

immutable

An immutable object's data cannot be changed.

importing

A means to allow the use of abbreviated class names.

index

A unique identifying integer for a component of an array.

inheritance

The mechanism by which a class has the data and methods of its parent classes.

instance variables

Non-static variables of a class.

integer

Any data type that represents non-fractional numbers.

interface

A list of public method declarations.

interpreted compiled language

A language whose compiled code is executed by a virtual machine.

Team LIB

◀ PREVIOUS

NEXT ▶

J

Java Virtual Machine

A virtual computer that runs Java programs.

Team LIB

◀ PREVIOUS

NEXT ▶

L

label

A name associated with a loop. Labels may be used with `break` and `continue` statements.

layout manager

Objects responsible for setting the location and size of components in a container.

listener

An object that should be notified when a component's state changes.

literal string

Text enclosed in double quotes.

look and feel

A GUI-based program's appearance (look) and responses to user input (feel).

loop

A piece of code that's executed repeatedly. The number of repetitions can be preset, or execution can continue until a condition is met.

loop counter

A variable that regulates the number of passes through a loop.

M

maintenance

The process of fixing bugs and adding features.

main thread

The thread that executes an application's `main()` method.

memory

A circuit that stores a digital value.

method caller

The code that calls a method.

modal dialog

A dialog that consumes all mouse and keyboard input.

modulo

An operation that divides the first operand by the second operand and returns the remainder. Its symbol is the `%` sign.

multidimensional

A term used to describe an array with components specified by more than one index.

multithreaded

Capable of performing more than one task at a time.

mutator/setter

Method used to support data hiding. It has a *void* return type and a single argument. By convention, the name of a mutator begins with `set`, followed by the property to be modified.

Team LIB

← PREVIOUS

NEXT →

N

namespace

A way of organizing resources (files, classes, etc.) so that name uniqueness has to be maintained only in relatively small and manageable regions.

nesting

The technique of putting a loop within a loop.

no-args constructor

A constructor with an empty argument list.

Team LIB

← PREVIOUS

NEXT →

O

object equality

An equality criterion that is true if two distinct objects have equal data.

objects

Objects are an individual instance of a class.

one-dimensional

A term used to describe an array with components specified by a single unique index.

operands

The values on which operators operate.

origin

The point with coordinates (0, 0) in a component; the upper-left corner.

overloading

Reuse of a method name in a class.

overriding

Reuse of a method name in an inheritance hierarchy.

P

package

A named group of interrelated classes.

pixel

An abbreviation for *picture element*. A single dot on a computer screen.

precedence

The order of execution when multiple operations are combined into a single statement.

preferred size

The default size of a component, usually derived from its label and font. Layout managers may honor or ignore preferred size.

primitives

The non-object data types: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.

private access

The most restrictive access mode. A private feature may be accessed only by an instance of the class that defines the feature.

protected access

An access mode that grants access to classes in the same package as, and subclasses of, the class that defines the feature.

public access

Completely unrestricted access.

R

radio button

A member of a group, only one of which can be selected at any time.

reader

A class that reads 8-bit text and delivers Unicode characters.

reference

A variable that exists in accessible memory and accesses an object or array in inaccessible memory.

reference equality

An equality criterion that is true if two references point to the same object.

return value

The value returned by a method.

row major

Specification of row followed by column.

S

scalable

Useful and efficient when usage or requirements increase.

scientific notation

A useful representation for expressing very large or very small numbers. The letter E is used as shorthand for "times ten to the...".

scope

A variable's scope is the matching pair of open and closed curly braces that most tightly encloses the variable's declaration.

serifs

Small decorations on the tips of letters that improve readability in medium to large fonts.

shifting

One of several operations that move the bits of an integral operand to the left or right by a certain number of positions.

short-circuit operator

An operator that does not evaluate its second operand if the value of the first operand is enough to determine the value of the operation.

signed

Supporting both positive and negative integer types.

side effect

A change in program state as a result of a method call.

source code

Code that must be translated into appropriate binary values before it can be executed by a computer.

stack trace

A listing of an application's method call hierarchy at the moment an exception was thrown.

static

Associated with a class, rather than with an individual instance of a class.

string concatenation

The consecutive joining of strings, one after another.

subclass

A class that extends a superclass, inheriting its data and methods.

subtractive primary colors

The primary colors of paints and dyes (red, yellow, and blue). They combine to form green, orange, and purple.

superclass

A class from which a subclass inherits data and methods.

T

ternary operator

An operator that takes three operands. Java's only ternary operator is `? : .`

thread

A single task in a multithreaded program.

throw

To interrupt normal program flow by raising an exception.

truncate

To discard the fractional part of a number.

try block

Code following the `try` keyword, from which exceptions might be thrown.

two's complement

A format used to represent signed integers.

Team LIB

← PREVIOUS

NEXT →

U

unary operators

Symbols that perform operations on a single operand.

unicode

A standard for associating characters of many alphabets with 16-bit data.

update

The final part of a for loop.

UTF

A standard for converting Unicode strings into bytes.

Team LIB

← PREVIOUS

NEXT →

Team LIB

◀ PREVIOUS

NEXT ▶

V

variable-width font

Font in which different characters have different widths.

virtual computer

An imaginary computer that is simulated on a real computer.

Team LIB

◀ PREVIOUS

NEXT ▶

W

white space

Blank space in source code, ignored by the compiler but useful in creating code that is more readable.

wrapper

A class whose data is a single primitive value. Java's eight wrapper classes are in the `java.lang` package.

writer

A class that reads Unicode characters and delivers 8-bit text.

Index

Note to the Reader: Throughout this index **boldfaced** page numbers indicate primary discussions of a topic. *Italicized* page numbers indicate illustrations.

A

- abstract classes and methods
 - defined, [468](#)
 - working with, [182–185](#), [183](#)
- access control, [172–174](#)
 - abstract modifier, [182–185](#), [183](#)
 - default access, [174–176](#)
 - final modifier, [180–182](#)
 - with overriding, [178–180](#), [179](#)
 - private access, [174–175](#)
 - protected access, [177–178](#)
 - public access, [174](#)
- access modifiers, [173–174](#), [468](#)
- AccessExample class, [173–174](#)
- accessible memory, [109–111](#), [110](#)
- accessors
 - with data hiding, [173](#)
 - defined, [468](#)
- action listeners, [333–339](#), [335–339](#)
- ActionListener interfaces, [334](#), [363](#)
- actionPerformed method
 - in ActionListener, [334](#)
 - in DisablingNim, [350](#)
 - in FancySrcCanvas, [390–391](#)
 - in FileDialogPractice, [367–368](#)
 - in GraphicOutputNim, [347–348](#)
 - in ListeningFrame, [341](#)
 - in MenuTest, [364–365](#)
 - in Simple Event Lab, [339](#)
 - in SimpleActionListener, [334](#)
 - in SimpleNim, [342–344](#)
 - source of, [339–340](#)
 - in TextAreaNim, [344–345](#)
- add method
 - for border layout managers, [318](#)
 - for buttons, [294](#)
 - for choices, [303](#)
 - for menu items, [307](#), [355](#)
 - for panels, [321](#)
- ADD opcode, [7](#)
- addActionListener method
 - for buttons, [334–335](#)
 - for menus, [363](#)
 - for scrollbars, [355](#)
- addAdjustmentListener method, [355](#)
- addItemListener method
 - for check boxes, [352](#)
 - for choices, [352](#), [376](#)
- addition
 - basic operator for, [37](#)
 - increment operator for, [45–46](#)
- additive primary colors
 - combining, [273–274](#)
 - defined, [468](#)
- addresses of bytes, [4](#), [4](#)
 - in instructions, [16](#)
 - vs. names, [26](#)
 - in opcodes, [7](#)
 - in SimCom, [9–10](#), [9](#)
 - start of, [5](#), [61](#)
 - vs. values, [12](#), [110](#)
- addSeparator method, [307](#)
- addTextListener method, [353](#)
- AdjustmentListener interface, [355](#)

- adjustments, events from, [355–356](#)
- adjustmentValueChanged method, [355–356](#)
- Advanced Exception Lab, [215](#), [216–217](#), [217](#)
- ageInNYears method, [126–127](#), [131](#)
- allocating memory
 - for arrays, [110](#)
 - defined, [468](#)
- American Standard Code for Information Interchange (ASCII)
 - defined, [468](#)
 - for file characters, [263](#)
- ampersands (&)
 - for bitwise operator, [40–41](#)
 - for boolean operator, [46–48](#)
 - in compound assignment, [51](#)
 - as short-circuit operator, [49](#)
- analog circuits
 - defined, [468](#)
 - uses for, [3](#)
- and operators
 - bitwise, [40–41](#)
 - boolean, [46–48](#), [47–48](#)
- AnimatedIllustrations directory, [398](#), [403](#)
- anonymous instances, [289](#)
- api directory, [224](#)
- API pages
 - downloading, [396](#), [403](#)
 - purpose of, [222–223](#)
 - structure of, [224–228](#), [224–227](#)
- append method
 - for text areas, [312](#), [348](#)
 - in TextArea, [344](#)
- Apple Developer Connection site, [401](#)
- applications, [468](#)
- argument bits, [6](#), [6](#)
- arguments
 - command-line, [235–236](#), [235](#)
 - for methods, [60–64](#), [62–63](#)
 - names for, [68](#)
- arithmetic operations
 - basic, [37–38](#)
 - bitwise, [40–41](#), [40–41](#)
 - modulo, [42](#)
 - precedence in, [38–40](#), [39](#)
 - shifting, [42–44](#), [42–44](#)
 - unary, [44–46](#)
- ArrayIndexOutOfBoundsException class, [205–208](#)
- arrays
 - creating, [103–104](#), [103](#)
 - declaring, [102](#)
 - defined, [468](#)
 - exercise questions for, [115–116](#)
 - exercise solutions for, [421–423](#)
 - garbage collection for, [114–115](#)
 - indices for, [102](#), [104](#), [104](#), [470](#)
 - initializing, [103](#), [105](#)
 - length of, [104–105](#)
 - loops for, [105–106](#)
 - multidimensional, [106–108](#), [106](#), [108–109](#)
 - as objects, [109–112](#), [110–112](#)
 - vs. objects, [118–119](#)
 - passing references to, [112–114](#), [113](#)
- ASCII (American Standard Code for Information Interchange)
 - defined, [468](#)
 - for file characters, [263](#)
- assemblers
 - code in, [8](#), [17](#)
 - defined, [468](#)
- assembly languages
 - code in, [8](#), [16](#), [17](#)
 - vs. compiled, [17–18](#), [18](#)
 - defined, [468](#)
- assignment operations, [16](#)
 - compound, [51](#)

operator for, [38](#)
process, [27–28](#)
asterisks (*)
 for comments, [36](#)
 in compound assignment, [51](#)
 with import, [171](#)
 for multiplication, [37](#)
AWT toolkit, [271](#)

Team LIB

◀ PREVIOUS NEXT ▶

Index

B

- background color for frames, [271](#), [275–276](#)
- backslashes (/) in filenames, [249–250](#)
- backward compatibility, [228](#)
- BarAndTF class, [356](#)
- BarAtNorth class, [317](#)
- BarChart class, [183](#), [185](#)
- bars (|)
 - for bitwise operator, [40–41](#)
 - for boolean operator, [46–48](#)
 - in compound assignment, [51](#)
 - for short-circuit operator, [49](#)
- base-2 code, [17](#)
- base-2 notation, [21](#)
- baselines
 - defined, [468](#)
 - for text, [283](#), [283](#)
- batch files, [398](#)
- bin directory, [397](#), [397](#)
- binary operators
 - defined, [468](#)
 - symbols for, [37](#)
- bits
 - defined, [468](#)
 - in memory, [3](#)
 - opcode and argument, [6](#), [6](#)
- bitwise operations
 - defined, [468](#)
 - process, [40–41](#), [40–41](#)
 - right-shift, [43–44](#), [43](#)
- BlackLineOnWhite class, [278](#)
- blocks
 - catch. See [catch blocks](#)
 - defined, [468](#)
 - for if statements, [74](#)
 - scope in, [68](#)
- blue color, [273–274](#)
- BlueRect class, [279](#)
- Boats class, [297–298](#)
- body of methods, [59](#)
- bold font style, [284–285](#)
- BoolArrayLab animated illustration, [108](#), [108–109](#)
- Boolean class, [226](#)
- boolean data type
 - defined, [468](#)
 - for if statements, [74](#)
 - for logical values, [25](#)
 - wrapper class for, [240](#)
- boolean operations
 - comparison, [50–51](#)
 - evaluation of, [46–48](#), [47–48](#)
 - short-circuit, [49–50](#)
- BooleanOps class, [46–47](#)
- BoolLab animated illustration, [47–49](#), [47–48](#)
- border layout managers, [313](#), [317–320](#), [317–319](#)
- bounding boxes
 - defined, [468](#)
 - for ovals, [280–281](#), [281](#)
- BoxLayout layout manager, [325](#)
- break statements
 - labeled, [94–97](#)
 - in loops, [88–89](#)
 - in switch statements, [79–81](#)

- breaking out of loops, [88–89](#)
- BtnInAFrame class, [294](#)
- bugs, finding, [206](#)
- buildColorChoice method, [370](#), [373](#)
- Button class, [293](#)
- buttons
 - in flow layout managers, [316](#)
 - working with, [293–295](#), [293](#), [295](#)
- byte data type
 - defined, [468](#)
 - range of, [21](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- bytecode
 - defined, [468](#)
 - in JVM, [19](#)
- bytes, [3](#), [3](#)
 - addresses of, [4](#), [4](#), [12](#)
 - on disks, [248–249](#)
 - reading, [249](#), [254–255](#), [256](#)
 - writing, [249](#), [251–252](#), [253](#)

Index

C

- callers for methods, [60](#), [471](#)
- calling methods, [60](#), [66–67](#)
- Canvas class, [377](#)
- CardLayout manager, [325](#)
- caret (^)
 - for bitwise operator, [40–41](#)
 - for boolean operator, [46–48](#)
 - in compound assignment, [51](#)
- Cartesian coordinates, [277](#), [278](#)
- case statements, [79](#)
- catch blocks
 - defined, [469](#)
 - execution in, [202–203](#)
 - with instanceof, [212–216](#), [216](#)
 - multiple, [210–212](#)
 - safety net, [213](#)
- catching exceptions, [202–203](#)
- CboxAndChoice class, [352–353](#)
- CboxInnaFrame class, [296](#)
- Center region, [317–319](#), [319](#)
- CenteredOval class, [282](#), [283](#)
- chaining
 - defined, [469](#)
 - input, [259](#), [259](#)
 - output, [256](#), [257](#)
- chains of constructions
 - defined, [469](#)
 - in inheritance, [149–152](#), [149](#)
- char data type
 - with result types, [52–53](#), [53](#)
 - for text, [25](#)
 - wrapper class for, [240](#)
- character code for files, [263](#)
- characters, [25](#)
- charAt method, [233](#)
- charIndexToX method, [383](#), [386](#)
- Chart class, [182–185](#), [183](#)
- checkboxes
 - events from, [351–353](#), [351](#)
 - in flow layout managers, [316](#)
 - working with, [296–300](#), [296–298](#), [300](#)
- CheckboxGroup class, [299](#)
- checked exceptions, [205](#)
 - with stack traces, [216–217](#)
 - throwing, [217–220](#)
 - working with, [208–209](#)
- CheckedCbox class, [297](#)
- Choice class, [302](#)
- choices
 - events from, [351–353](#), [353](#)
 - working with, [301–304](#), [301–302](#), [304](#)
- ChooseFontByRadios class, [301](#)
- circles, [280](#)
- clamp method, [75](#)
- class definitions, [35](#), [120](#)
- class files
 - compiler output, [19](#), [29](#)
 - defined, [469](#)
- class keyword, [119](#)
- class loaders
 - defined, [469](#)

- functions of, [132–133](#), [170](#)
- .class suffix, [19](#), [29](#)
- classes, [119–120](#), [119](#)
 - abstract, [182–185](#), [183](#)
 - core. See [core classes and packages](#)
 - in packages, [165](#)
- classname-dot-staticVariableName syntax, [130](#)
- classpath elements, [168–170](#), [469](#)
- CLASSPATH environment variable
 - for executables
 - in Macintosh, [402–403](#)
 - in Windows, [396](#), [398–399](#)
 - for packages, [169](#)
- classpath option in javac, [169](#)
- close method, [251](#), [258](#)
- closing streams, [251](#)
- colons (:)
 - in classpath elements, [169](#)
 - for labels, [96](#)
 - in ternary operator, [77](#)
- color
 - in final project, [368–376](#), [369](#), [372](#), [378–389](#), [379](#)
 - for frames, [271](#), [275–276](#)
 - for painting, [273–276](#), [275–276](#)
- Color class, [183](#), [237–238](#), [274](#)
- Color Lab program, [275–276](#), [275–276](#)
- ColorChoice class, [374–375](#)
- ColorChoiceTest class, [374–375](#)
- ColorTest class, [372–374](#)
- columns in text areas, [310](#)
- command-line arguments, [235–236](#), [235](#)
- comments
 - defined, [469](#)
 - in Frame Lab, [287](#)
 - painting, [383–384](#)
 - types of, [36](#)
- comparison operators, [50–51](#)
- compatibility, backward, [228](#)
- compiled languages, [16–17](#)
 - vs. assembly, [17–18](#), [18](#)
 - defined, [469](#)
- compiler
 - downloading, [396](#)
 - for packages, [166–167](#), [168](#), [169](#)
 - references with, [155](#)
- compiling, [206](#)
- components, [102](#), [292](#), [292](#)
 - buttons, [293–295](#), [293](#), [295](#)
 - checkboxes
 - events from, [351–353](#), [351](#)
 - in flow layout managers, [316](#)
 - working with, [296–300](#), [296–298](#), [300](#)
 - choices
 - events from, [351–353](#), [353](#)
 - working with, [301–304](#), [301–302](#), [304](#)
 - defined, [469](#)
 - events for. See [events](#)
 - exercise questions for, [327–328](#)
 - exercise solutions for, [448–454](#)
 - labels, [304–305](#), [305](#)
 - layout managers. See [layout managers](#)
 - menus, [305–309](#), [307–308](#)
 - scrollbars, [312–313](#), [313](#)
 - text areas, [310–312](#), [311–312](#)
 - text fields, [309–310](#), [310](#)
- compound assignments, [51](#)
- computePixel method, [90](#)
- concat method, [233](#)
- concatenation of strings, [233](#), [237–239](#), [238–239](#)
- ConcatLab animated illustration, [238–239](#), [238–239](#)

- conditionals, [74](#)
 - defined, [469](#)
 - exercise questions for, [98–99](#)
 - exercise solutions for, [416–420](#)
 - in for loops, [87, 87](#)
 - if statements, [74–76](#)
 - switch statement, [77–81](#)
 - ternary operator, [76–77](#)
- ConnectException class, [210–211, 213–215, 219](#)
- constants
 - benefits of, [181–182](#)
 - in interfaces, [193–194](#)
- construction
 - chains of, [149–152, 149](#)
 - defined, [469](#)
 - with new, [111](#)
- ConstructorLab animated illustration, [150–152](#)
- constructors, [146–147](#)
 - in API pages, [227, 227](#)
 - default, [148–149](#)
 - defined, [469](#)
 - overloading, [147–148](#)
- Container class, [313](#)
- containers, [469](#)
- contexts, graphics, [277](#)
- continue statement
 - labeled, [94–97](#)
 - purpose of, [89–90](#)
- coordinates, [277, 278](#)
- core classes and packages, [205, 222–223](#)
 - API pages for, [222–228, 224–227](#)
 - exercise questions for, [244–246](#)
 - exercise solutions for, [440–442](#)
 - java.lang, [228](#)
 - java.lang.Integer, [240–241](#)
 - java.lang.Math, [243–244](#)
 - java.lang.Object, [236–239, 238–239](#)
 - java.lang.String, [229–236, 231–232, 234–235](#)
 - java.lang.System, [241–243](#)
- cos method, [243](#)
- cp option in javac, [169](#)
- CreateArrayLab animated illusion, [111–113, 113](#)
- .cshrc file for paths, [402](#)
- curly brackets ({})
 - for arrays, [105](#)
 - for constructors, [146](#)
 - for definitions, [35](#)
 - for do-while loops, [86](#)
 - in for loops, [91](#)
 - for if statements, [74](#)
 - for interfaces, [188](#)
 - for method declarations, [59](#)
 - for scope, [68](#)
 - for while loops, [82](#)
- cycloids, [91–92, 91](#)

Index

D

- d option in package, [166–167](#)
- data and data types, [16, 19](#)
 - boolean, [25](#)
 - characters, [25](#)
 - in declarations, [27](#)
 - declaring and assigning, [26–28](#)
 - exercise questions for, [30–31](#)
 - exercise solutions for, [407–409](#)
 - floating-point, [24–25](#)
 - integer, [21–24, 22–23](#)
 - in interfaces, [192–194](#)
 - for objects, [120–122, 121–122](#)
 - summary, [26](#)
- Data Chain Lab animated illustration, [261, 262](#)
- data hiding
 - defined, [469](#)
 - in object-oriented programming, [172–173](#)
- DataInputStream class, [256, 259](#)
- DataLab animated illustration, [122, 122](#)
- DataOutputStream class, [256–257](#)
- debug code, [469](#)
- declarations, [16, 26–28](#)
 - for arrays, [102](#)
 - defined, [469](#)
 - in interfaces, [188–189](#)
 - for methods, [59](#)
- decrement operator, [45–46](#)
- default access
 - defined, [469](#)
 - purpose of, [174–176](#)
- default code, [81](#)
- default constructors
 - defined, [469](#)
 - purpose of, [148–149](#)
- default statements, [79–80](#)
- definitions, class, [35, 120](#)
- deprecated methods
 - compatibility of, [228](#)
 - defined, [469](#)
- destination directories
 - defined, [469](#)
 - for packages, [167](#)
- Developer Tools package, [402](#)
- Developer Tools Update, [401–402](#)
- dialog boxes
 - class for, [365–366, 366](#)
 - defined, [469](#)
- differences, [37](#)
- digital circuits, [2](#)
 - vs. analog, [3](#)
 - defined, [469](#)
- digital computers, [2](#)
 - vs. analog, [3](#)
 - defined, [470](#)
- Dimension class, [282](#)
- dimensions for arrays, [106–108, 106, 108–109](#)
- directories, [165](#)
 - for installation files, [397, 403](#)
 - for packages, [166–167, 166](#)
 - for programs, [398](#)
- disabling components, [349](#)
- DisablingNim class, [349–351](#)
- disks, [248–249](#). *See also* [files](#)

- display method, [183–185](#)
- division
 - operator for, [37](#)
 - truncation with, [40](#)
- do-while loops, [85–86](#)
- docs directory, [224](#)
- Dog class, [120](#)
- double data type
 - defined, [470](#)
 - range of, [24–25](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- double-quotes (") for literal strings, [30](#), [229](#)
- downloading and installing Java
 - in Macintosh, [401–404](#)
 - overview, [396](#)
 - in Windows, [396–401](#), [397](#)
- drawing, [277](#), [278](#). See also [painting](#)
 - circles, [280](#)
 - filling in, [281–282](#), [281](#), [283](#)
 - frames, [287–289](#), [287–289](#)
 - lines, [278](#), [279](#)
 - ovals, [280–281](#), [280–281](#)
 - rectangles, [279](#), [279](#)
 - squares, [280](#)
 - text, [283–286](#), [284–286](#)
- drawLine method, [278](#)
- drawOval method, [280–281](#)
- drawRect method, [279–280](#)
- drawString method, [283–285](#)
- dump method, [129](#)
- dumpSalary method, [146](#)

Index

E

- E variable, [244](#)
- earnMoreThan method, [175](#)
- East region, [317–319](#), [318–319](#)
- Edit menus, guidelines for, [308](#)
- editors for source files, [400](#), [404](#)
- ellipsis (...)
 - for button labels, [336](#)
 - defined, [470](#)
 - for dialog boxes, [309](#)
- else statement, [74–75](#)
- else if statement, [76](#)
- Employee class
 - overriding in, [179](#), [179](#)
 - private access in, [174–175](#)
 - as superclass, [142–143](#), [143](#), [145](#)
- empty strings, [229](#), [470](#)
- EmptyFrame class, [271–272](#)
- environment variable, [169](#)
- equal signs (=)
 - in arithmetic operations, [38](#)
 - for assignment, [27](#)
 - for comparisons, [50](#)
 - for reference equality, [234](#)
- equality
 - object, [234](#), [234](#), [236](#)
 - reference, [234](#), [234](#)
- equals method
 - in Object, [236](#)
 - in Point, [236](#)
 - in String, [233–234](#)
- equalsIgnoreCase method, [233](#)
- error codes and messages, [27](#), [198–200](#)
- escape codes, [28](#)
- EvaluatorLab animated illustration, [39–40](#), [39](#)
- event dispatch threads, [332–333](#)
- event-driven programs, [330–332](#), [331](#)
 - defined, [470](#)
 - threads in, [332–333](#)
- Event Lab animated illustration, [354–355](#), [354](#)
- events, [330](#)
 - actions for, [333–339](#), [335–339](#)
 - from checkboxes, choices, and items, [351–353](#), [351](#), [353](#)
 - defined, [470](#)
 - exercise questions for, [357–358](#)
 - exercise solutions for, [454–460](#)
 - information from, [339–343](#), [340](#), [342](#)
 - from menus, [355](#)
 - in Nim game, [342–351](#), [342–344](#), [346](#), [349](#)
 - from scrollbars and adjustments, [355–356](#), [356](#)
 - from text fields and text areas, [353–355](#), [354–355](#)
- Exception class, [200](#), [205](#), [215](#)
- exceptions, [198](#)
 - catching, [202–203](#)
 - checked, [205](#)
 - with stack traces, [216–217](#)
 - throwing, [217–220](#)
 - working with, [208–209](#)
 - defined, [470](#)
 - exercise questions for, [220](#)
 - exercise solutions for, [438–440](#)
 - families of, [205–206](#)
 - real world, [203](#)
 - runtime, [205–208](#)
 - throwing, [200–201](#), [217–220](#)

- exclamation points (!)
 - for comparisons, [50](#)
 - for inversion, [46–48](#)
- exclusive or operators
 - bitwise, [40–41](#)
 - boolean, [46–48](#), [47–48](#)
- executing bytes, [5](#)
- exit codes, [242](#)
- exit method, [242](#)
- exponents in scientific notation, [24](#)
- expressions in switch statements, [78](#)
- extending interfaces, [194–195](#)
- extends keyword, [142–143](#), [194](#)

Team LIB

← PREVIOUS

NEXT →

Index

F

- falling through switch statements
 - bugs from, [81](#)
 - defined, [470](#)
- false value, [25](#)
- families of fonts, [205–206](#), [284](#)
- FancyButtonInFrame class, [295](#)
- fancysrc package, [389](#)
- FancySrcCanvas class, [377–391](#)
- FancySrcFrame class, [361–362](#), [361](#), [371](#)
- fields
 - in API pages, [227](#), [227](#)
 - defined, [470](#)
 - for objects, [121–122](#)
 - text fields, [309–310](#), [310](#)
 - events from, [353–355](#), [354–355](#)
 - in flow layout managers, [316](#)
- File menu
 - in final project, [362–365](#), [362](#), [364](#)
 - guidelines for, [308](#)
- file separators
 - defined, [470](#)
 - problems with, [249](#)
- FileDialog class, [365–366](#), [366](#)
- FileDialogPractice class, [367–368](#)
- FileInputStream class, [249](#), [254](#)
- filenames, backslashes in, [249–250](#)
- FileNotFoundException class, [251–252](#)
- FileOutputStream class, [249](#), [251](#)
- FileReader class, [263](#), [265](#)
- files
 - character code for, [263](#)
 - Data Chain Lab for, [261](#), [262](#)
 - exercise questions for, [267](#)
 - exercise solutions for, [443–445](#)
 - in final project, [365–368](#), [366](#)
 - line number readers for, [265](#), [266](#)
 - names for, [249–250](#)
 - new lines in, [264–265](#)
 - reading, [249](#)
 - bytes, [254–255](#), [256](#)
 - data, [259–261](#), [259](#), [262](#)
 - as sequences of bytes, [248–249](#)
 - writing, [249](#)
 - bytes, [251–252](#), [253](#)
 - data, [256–259](#), [257](#)
- FileWriter class, [263–264](#)
- Filled class, [281](#)
- filling in drawing, [281–282](#), [281](#), [283](#)
- fillRect method, [281](#)
- final modifier
 - defined, [470](#)
 - working with, [180–182](#)
- final project
 - colors in, [368–376](#), [369](#), [372](#)
 - description of, [360–362](#), [360–361](#)
 - exercise questions for, [392–393](#)
 - exercise solutions for, [461–465](#)
 - File menu in, [362–365](#), [362](#), [364](#)
 - main display area in, [376–378](#), [377](#)
 - painting in, [378–389](#), [379](#)
 - parts of
 - building, [361–362](#)
 - combining, [389–391](#)
 - specifying files in, [365–368](#), [366](#)

- finding packages, [168–170](#)
- Fish class, [177](#)
- flags
 - defined, [470](#)
 - for program status, [95](#)
- float data type
 - defined, [470](#)
 - range of, [24–25](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- floating-point data types, [24–25](#)
- Floating-Point Lab animated illustration, [25](#)
- Flow Lab animated illustration, [316](#), [316](#)
- flow layout managers, [313–316](#), [315–316](#)
- FlowLayout class, [294](#)
- Font class, [285](#)
- Font Lab program, [286](#), [286](#)
- FontAndBaseline class, [283–284](#)
- FontChoice class, [303](#)
- FontChoiceWithLabels class, [304–305](#)
- fonts
 - choices for, [301–304](#), [301–302](#), [304](#)
 - drawing, [283–286](#), [284–286](#)
- for loops
 - for arrays, [105–106](#)
 - structure of, [86–89](#), [87](#)
 - variables in, [97](#)
- fractions, [24](#)
- Frame class, [271](#)
- Frame Lab animated illustration, [287–289](#), [287–289](#)
- frames
 - color for, [271](#), [275–276](#)
 - drawing, [287–289](#), [287–289](#)
 - in painting, [270–273](#), [270](#)
 - text in, [284–285](#), [284–285](#)
- FrameWithSimpleMenu class, [306](#)
- FrameWithSubmenu class, [307–308](#)
- friendly access, [174](#)

Index

G

- garbage collection
 - defined, [470](#)
 - purpose of, [114–115](#)
 - threads for, [332](#)
- garbage collection threads
 - defined, [470](#)
 - purpose of, [332](#)
- getAvailableFontFamilyNames method, [286](#)
- getAverageTemp method, [135–136](#)
- getColorFromChoice method, [371–373](#)
- getDirectory method, [367](#)
- getFile method, [367–368](#)
- getLineNumber method, [265](#)
- getLocalGraphicsEnvironment method, [286](#)
- getMass method, [65–66](#)
- getMessage method, [201–202](#), [219](#)
- getNumEnvelopesInStock method, [210–211](#), [218–220](#)
- getRainfall method
 - error codes for, [198–200](#)
 - exceptions for, [200–203](#)
- getSalary method, [174–175](#)
- getSelectedColor method, [374–375](#)
- getSelectedIndex method, [353](#), [374](#)
- getSize method
 - in Canvas, [380](#)
 - in Frame, [282](#)
- getSource method
 - in ActionEvent, [340](#)
 - in ItemEvent, [352–353](#)
- getState method, [353](#)
- getters, [173](#)
- getValue method, [355](#)
- getWeightKg method, [180–181](#)
- getWeightLbs method, [181](#)
- graphical user interface (GUI)
 - classes for, [228](#)
 - events in. See [events](#)
 - painting in. See [painting](#)
- GraphicOutputNim class, [346–347](#)
- Graphics class, [277–282](#), [278–281](#)
- graphics contexts, [277](#)
- graphics objects, [277](#)
- GraphicsEnvironment class, [286](#)
- greater than signs (>)
 - for comparisons, [50](#)
 - in compound assignment, [51](#)
 - in shifting operations, [42](#)
- green color, [273–274](#)
- GridBagLayout manager, [325](#)
- GridLayout manager, [325](#)
- GUI (graphical user interface)
 - classes for, [228](#)
 - events in. See [events](#)
 - painting in. See [painting](#)
- GUI threads
 - defined, [470](#)
 - in JVM, [332–333](#)

Index

H

HALT opcode, [8](#)

height

of canvas, [380](#)

of dialog boxes, [367](#)

in text areas, [310](#)

Help menus, guidelines for, [309](#)

-help option in java, [236](#)

hiding data, [173](#)

horizontal scrollbars, [312–313](#)

howBig method, [76](#)

Index

I

- IDE (Integrated Development Environment)
 - in Macintosh, [404](#)
 - in Windows, [400](#)
- if statements, [74–76](#)
- immutable classes, [229](#)
- immutable objects, [470](#)
- implements keyword, [188](#)
- import statement, [171–172](#)
- importing
 - defined, [470](#)
 - packages, [170–172](#)
- in variable, [241–242](#)
- inaccessible memory, [109–111](#), [110](#)
- increment operator, [45–46](#)
- incrementing program counter, [7](#)
- indexOf method, [266–267](#), [382](#)
- indices
 - array, [102](#), [104](#), [104](#)
 - defined, [470](#)
- indirect addresses, [7](#)
- information from events, [339–343](#), [340](#), [342](#)
- Inherit Lab animated illustration, [144–145](#), [144–145](#)
- inheritance, [140–142](#)
 - with constructors, [146–152](#), [149](#)
 - defined, [470](#)
 - example, [145–146](#)
 - exercise questions for, [160–161](#)
 - exercise solutions for, [426–431](#)
 - with interfaces, [189–190](#), [189](#)
 - method overriding in, [152–153](#), [152](#)
 - polymorphism with, [154–160](#)
 - from superclasses, [142–145](#), [144–145](#)
- initialization
 - array, [103](#), [105](#)
 - in for loops, [87](#), [87](#), [97](#)
- input, file. See [files](#)
- instance variables, [130](#), [470](#)
- instanceof keyword
 - catch blocks with, [212–216](#), [216](#)
 - for references, [191–192](#)
- instruction sets, [18](#)
- int data type
 - ranges of, [21](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- integers
 - data types for, [21](#)
 - defined, [471](#)
 - two's complement format for, [21–24](#), [22–23](#)
- Integrated Development Environment (IDE)
 - in Macintosh, [404](#)
 - in Windows, [400](#)
- interfaces, [188](#)
 - data in, [192–194](#)
 - defined, [471](#)
 - exercise questions for, [195–196](#)
 - exercise solutions for, [431–435](#)
 - extending, [194–195](#)
 - method declarations in, [188–189](#)
 - objects and references in, [190–192](#)
- interpreted compiled languages
 - defined, [471](#)
 - Java as, [19](#)

- introductory material
 - exercise questions for, [13](#)
 - exercise solutions for, [406–407](#)
 - memory, [2–4](#), [3–4](#)
 - SimCom virtual computer, [5–12](#), [6](#), [9](#)
- inversion operator, [46–48](#)
- invoking methods, [60](#)
- IOException class, [214](#), [218](#), [251–252](#)
- italic font style, [284–285](#)
- item events, [351–353](#)
- ItemListener interface, [352](#)
- itemStateChanged method
 - in CboxAndChoice, [353](#)
 - in ColorChoiceTest, [375](#)
 - in ColorTest, [373–374](#)
 - in FancySrcFrame, [371](#)
 - in ItemListener, [351–352](#)

Index

J

- jar file
 - installing, [397–398](#), [397](#), [400](#)
 - running, [403](#)
- java.awt package, [228](#)
 - API pages for, [225](#)
 - components in, [292](#), [292](#)
- java.awt.Button class, [293](#)
- java.awt.Canvas class, [377](#)
- java.awt.CheckboxGroup class, [299](#)
- java.awt.Choice class, [302](#)
- java.awt.Color class, [183](#), [237–238](#), [274](#)
- java.awt.event.ActionListener interface, [334](#), [363](#)
- java.awt.event.AdjustmentListener interface, [355](#)
- java.awt.event.ItemListener interface, [352](#)
- java.awt.FileDialog class, [365–366](#), [366](#)
- java.awt.Frame class, [271](#)
- java.awt.Graphics class, [277–282](#), [278–281](#)
- java.awt.LayoutManager interface, [325](#)
- java.awt.Panel class, [320](#)
- java.awt.Point class, [236](#)
- java.awt.Scrollbar class, [312](#)
- java.io package, [249](#), [256](#)
- java.lang package, [226](#), [228](#)
- java.lang.Boolean class, [226](#)
- java.lang.Integer class, [240–241](#)
- java.lang.Math class, [243–244](#)
- java.lang.Object class, [236–239](#), [238–239](#)
- java.lang.String class, [226](#), [229–236](#), [231–232](#), [234–235](#)
- java.lang.System class, [241–243](#)
- java.sql package, [228](#)
- java.util package, [228](#)
- Java Virtual Machine (JVM), [12](#), [19](#), [20](#)
 - class loaders in, [132–133](#)
 - defined, [471](#)
 - downloading, [396](#)
 - initialization in, [132](#)
- javac compiler
 - classpath option in, [169](#)
 - directory option in, [167](#)
 - installing, [397–398](#), [397](#)
- javax.swing package, [325](#)
- joining strings, [233](#), [237–239](#), [238–239](#)
- JRE (Java Runtime Environment), [397](#)
- JUMP opcode, [7](#)
- JUMPZ opcode, [7–8](#)
- JVM (Java Virtual Machine), [12](#), [19](#), [20](#)
 - class loaders in, [132–133](#)
 - defined, [471](#)
 - downloading, [396](#)
 - initialization in, [132](#)

Team LIB

← PREVIOUS

NEXT →

Index

K

keywordChoice class, [369](#)

killing frames, [272](#)

Team LIB

← PREVIOUS

NEXT →

Index

L

- L for long data type, [54](#)
- labels
 - defined, [471](#)
 - in flow layout managers, [316](#)
 - for loops, [94–97](#)
 - working with, [304–305](#), [305](#)
- Layout Lab animated illustration, [322–324](#), [322–324](#), [326](#), [326](#)
- layout managers, [294](#), [313–314](#)
 - border, [317–320](#), [317–319](#)
 - CardLayout, GridLayout, and GridBagLayout, [325](#), [326](#)
 - defined, [471](#)
 - flow, [314–316](#), [315–316](#)
 - lab for, [322–324](#), [322–324](#)
 - panels for, [320–324](#), [320](#), [322–324](#)
- LayoutManager interface, [325](#)
- LEFT area, [315](#)
- left-shift operation, [42](#), [42](#)
- length
 - of arrays, [104–105](#)
 - of strings, [233](#)
- length method, [233](#)
- less than signs (<)
 - for comparisons, [50](#)
 - in compound assignment, [51](#)
 - in shifting operations, [42](#)
- license agreements, [397](#), [399](#)
- LineNumberReader class, [265](#), [266](#)
- lines, drawing, [278](#), [279](#)
- listeners
 - defined, [471](#)
 - for events, [333–339](#), [335–339](#)
- ListeningFrame class, [341](#)
- literal strings, [30](#)
 - defined, [471](#)
 - for string instances, [229](#)
- LOAD opcode, [7–8](#)
- logical values, [25](#)
- .login file, [402](#)
- long data type
 - range of, [21](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- look and feel of programs
 - components for. See [components](#)
 - defined, [471](#)
- loop counters, [10](#)
 - defined, [471](#)
 - in for loops, [88](#)
- loops, [10](#), [82](#)
 - for arrays, [105–106](#)
 - breaking out of, [88–89](#)
 - continue statement in, [89–90](#)
 - defined, [471](#)
 - do-while, [85–86](#)
 - exercise questions for, [98–99](#)
 - exercise solutions for, [416–420](#)
 - for, [86–89](#), [87](#)
 - labels for, [94–97](#)
 - nesting, [90–94](#), [91–94](#)
 - scope in, [97](#)
 - while, [82–86](#), [84–85](#)
- lower case characters, [230–232](#), [231–232](#)

Index

M

Macintosh computers, downloading and installing Java on, [401–404](#)

main display area in final projects, [376–378](#), [377](#)

main method, [132–134](#), [133](#)

main threads
defined, [471](#)
purpose of, [332](#)

maintenance
and code duplication, [141](#)
defined, [471](#)

Manager class, [140–141](#), [143](#), [143](#), [151–152](#)

max method, [243](#)

MB prefix, [4](#)

mega prefix, [4](#)

memory, [2–3](#), [3](#)
for arrays, [109–111](#), [110](#)
defined, [471](#)
garbage collection for, [114–115](#)
organization of, [4](#), [4](#)
in SimCom, [6](#), [6](#)

memory leaks, [114](#)

menu bars, [362](#)

Menu class, [307](#)

menuListener, [355](#)

menus
events from, [355](#)
in final project, [362–365](#), [362](#), [364](#)
working with, [305–309](#), [307–308](#)

MenuTest class, [363–365](#)

method callers, [60](#), [471](#)

method definitions, [35](#)

MethodLab animated illustration, [61–64](#), [62](#)

methods, [58](#)
abstract, [182](#)
in API pages, [227](#), [227](#)
arguments for, [60–64](#), [62–63](#)
calling, [60](#), [66–67](#)
deprecated, [228](#)
exercise questions for, [70–71](#)
exercise solutions for, [413–415](#)
final, [180–182](#)
inheritance with, [143](#)
in interfaces, [188–189](#), [194](#)
main, [132–134](#), [133](#)
for objects, [126–127](#), [128](#)
order of execution, [68](#)
overriding, [152–153](#), [152](#)
polymorphism with, [65–66](#), [155–156](#)
references to, [112–114](#), [113](#)
return types for, [60–61](#), [64–65](#)
scope of, [68–69](#)
static, [130–132](#)
structure of, [58–61](#)

min method, [243](#)

minus signs (-)
in compound assignment, [51](#)
for subtraction, [37](#)
as unary operator, [44](#)

modal dialog boxes
characteristics of, [366](#)
defined, [471](#)

modulo operation
defined, [471](#)
operator for, [42](#)

Monospaced fonts, [285](#)

- multi-line comments, [36](#)
- multidimensional arrays
 - defined, [471](#)
 - working with, [106–108](#), [106](#), [108–109](#)
- multiple catch blocks, [210–212](#)
- multiple objects, [122–125](#), [123](#), [125](#)
- multiplication, [37](#)
- multithreaded devices
 - defined, [471](#)
 - JVM as, [332](#)
- mutators
 - with data hiding, [173](#)
 - defined, [471](#)

Team LIB

◀ PREVIOUS NEXT ▶

Index

N

- `\n` character, [264–265](#)
- n-dimensional arrays, [107](#)
- names
 - for arguments, [68](#)
 - for classes, [165](#)
 - for constructors, [146](#)
 - in declarations, [27](#)
 - for files, [249–250](#)
 - for memory locations, [26](#)
 - for methods, [60](#)
 - reusing, [154](#)
 - for variables, [68–69](#)
- namespaces
 - defined, [471](#)
 - directories for, [165](#), [165](#)
- nCubed method, [67](#)
- NEAndW class, [319](#)
- negative numbers, [21](#), [23](#)
- NestedLoopLab animated illustration, [91–94](#), [92–94](#)
- nesting
 - defined, [472](#)
 - if statements, [75](#)
 - loops, [90–94](#), [91–94](#)
 - menus, [308](#)
 - parentheses, [39](#)
- new keyword, [103](#)
- newline characters
 - in files, [264–265](#)
 - printing, [242](#)
 - for text areas, [312](#)
- Nim game, [342–351](#), [342–344](#), [346](#), [349](#)
- Nim Lab program, [343–351](#), [343–344](#), [346](#), [349](#)
- no-args constructors, [148](#)
 - defined, [472](#)
 - with superclasses, [150](#)
- NoMethods class, [58](#)
- North region, [317–319](#), [318–319](#)
- null value
 - with readline, [266](#)
 - with references, [112](#), [135–136](#)
- NullLayout class, [326](#)
- NumberFormatException class, [241](#)
- numeric operations, [52](#)

Index

O

- Object class, [236–239](#), [238–239](#)
- object-oriented programming, [119](#)
- ObjectLifeCycleLab animated illustration, [133–134](#), [133–134](#)
- ObjectMethodLab animated illustration, [127](#), [127](#)
- objects, [118](#)
 - vs. arrays, [118–119](#)
 - arrays as, [109–112](#), [110–112](#)
 - classes, [119–120](#), [119](#)
 - data for, [120–122](#), [121–122](#)
 - defined, [472](#)
 - equality of, [234](#), [234](#), [236](#), [472](#)
 - exercise questions for, [136–137](#)
 - exercise solutions for, [423–426](#)
 - methods for, [126–127](#), [128](#)
 - multiple, [122–125](#), [123](#), [125](#)
 - reference data with, [134–136](#)
 - references to, [121](#), [121](#), [190–192](#)
 - static data in, [128–130](#)
 - static methods in, [130–132](#)
- odometers, base-2, [21–22](#), [22](#)
- Officer class, [151–153](#)
- one-dimensional arrays
 - characteristics of, [106](#)
 - defined, [472](#)
- opcode bits, [6](#), [6](#)
- opcodes, [6–7](#)
- Open... menu item, [362](#)
- operands, [37](#), [472](#)
- operation codes, [6](#), [6](#)
- operations, [34](#)
 - arithmetic
 - basic, [37–38](#)
 - bitwise, [40–41](#), [40–41](#)
 - modulo, [42](#)
 - precedence in, [38–40](#), [39](#)
 - shifting, [42–44](#), [42–44](#)
 - unary, [44–46](#)
 - boolean
 - comparison, [50–51](#)
 - evaluation in, [46–48](#), [47–48](#)
 - short-circuit, [49–50](#)
 - comments, [36](#)
 - compound assignment, [51](#)
 - exercise questions for, [55–56](#)
 - exercise solutions for, [409–412](#)
 - result types in, [52–54](#), [53](#)
 - white space, [34](#)
- or operators
 - bitwise, [40–41](#)
 - boolean, [46–48](#), [47–48](#)
- order of method execution, [68](#)
- origins
 - defined, [472](#)
 - in drawing, [277](#), [278](#)
- OS X Developer Tools, [401](#)
- out variable, [241–242](#)
- output
 - file. See [files](#)
 - printing, [29–30](#)
- Oval class, [133](#)
- ovals
 - drawing, [280–281](#), [280–281](#)
 - filled, [281–282](#), [281](#)
- OverlayLayout layout managers, [325](#)

overloading
 constructors, [147–148](#)
 defined, [472](#)
 methods, [65](#)

overriding
 access control with, [178–180](#), [179](#)
 defined, [472](#)
 methods, [152–153](#), [152](#)

Team LIB

◀ PREVIOUS NEXT ▶

Index

P

- package access, [174](#)
- package keyword, [166–167](#)
- packages, [164](#), [164](#)
 - access control in. See [access control](#)
 - core. See [core classes and packages](#)
 - creating, [166–167](#), [166](#), [168](#)
 - defined, [472](#)
 - exercise questions for, [186](#)
 - exercise solutions for, [435–438](#)
 - finding, [168–170](#)
 - importing, [170–172](#)
 - interfaces in, [188](#)
 - and namespaces, [165–166](#), [165](#)
- paint method, [276–277](#)
 - in BlackLineOnWhite, [278](#)
 - in BlueRect, [279](#)
 - in CenteredOval, [282](#)
 - in ColorChoiceTest, [376](#)
 - in ColorTest, [373–374](#)
 - in DisablingNim, [350](#)
 - in FancySrcCanvas, [378](#), [380](#), [387](#)
 - in Filled, [281](#)
 - in FontAndBaseline, [284](#)
 - in Frame, [287](#)
 - in GraphicOutputNim, [347–348](#)
 - in ThreeOvals, [280](#)
 - in Xxx, [331](#)
- painting, [270](#)
 - color for, [273–276](#), [275–276](#)
 - drawing shapes. See [drawing](#)
 - exercise questions for, [290](#)
 - exercise solutions for, [445–448](#)
 - in final project, [378–389](#), [379](#)
 - Frame Lab for, [287–289](#), [287–289](#)
 - frames in, [270–273](#), [270](#)
 - process, [276–277](#)
 - text, [283–286](#), [284–286](#)
- paintLines method, [380](#), [387](#)
- paintOneSourceLine method, [382–384](#), [388–389](#)
- paintRegion method
 - in BarChart, [185](#)
 - in Chart, [185](#)
 - in PieChart, [185](#)
- paintText method, [380–382](#), [387–388](#)
- Panel class, [320](#)
- PanelInFrame class, [320–321](#)
- panels, [320–324](#), [320](#), [322–324](#)
- parabolas, [108](#), [109](#)
- parentheses ()
 - in arithmetic operations, [38–40](#), [39](#)
 - in boolean operations, [47](#)
 - in do-while loops, [86](#)
 - in for loops, [87](#)
 - for if statements, [74](#)
 - for methods, [60](#)
- parseInt method, [241](#)
- PartTimer class, [180](#)
- PassArrayLab animated illustration, [113–114](#)
- passing
 - arguments, [63](#), [67–68](#)
 - references to methods, [112–114](#), [113](#)
- PATH environment variable, [396](#), [398–399](#), [403](#)
- payEveryone method, [156–159](#)
- Paymaster class, [156–159](#), [180](#)
- percent signs (%)

- in compound assignment, [51](#)
 - in modulo operations, [42](#)
- periods (.) for object properties, [118](#)
- Person class, [120–123](#), [126–131](#)
- PI variable, [244](#)
- PieChart class, [183](#), [185](#)
- pixels
 - defined, [472](#)
 - for frames, [90](#), [272](#)
- plain font style, [284–285](#)
- plus signs (+)
 - for addition, [37](#)
 - in compound assignment, [51](#)
 - for string concatenation, [237–238](#)
 - as unary operator, [44](#)
- PlusPlusMinusMinus class, [45](#)
- Point class, [236](#)
- Point3D class, [239](#), [239](#)
- polymorphism
 - with inheritance, [154–160](#)
 - with methods, [65–66](#)
- position
 - in drawings, [277](#), [278](#)
 - of text, [283](#)
- post-decrement operator, [46](#)
- post-increment operator, [46](#)
- PostDec class, [46](#)
- pow method, [243](#)
- pre-decrement operator, [46](#)
- pre-increment operator, [46](#)
- precedence
 - in arithmetic operations, [38–40](#), [39](#)
 - in boolean operations, [47](#)
 - defined, [472](#)
 - summary, [54–55](#)
- preferred size
 - defined, [472](#)
 - with layout managers, [314](#), [317](#)
- primary colors, [273–274](#)
- primitive data types
 - defined, [472](#)
 - summary, [26](#)
- print2Cubes method, [66–67](#)
- print2Vals method, [66](#)
- print3x method, [67](#)
- printChars method, [233](#)
- printCheck method
 - in Employee, [142–143](#), [145–146](#), [179](#)
 - in Officer, [153](#)
 - overriding, [152–153](#)
 - in PartTimer, [180](#)
 - in Worker, [140–141](#)
- PrinterIOException class, [205](#), [209](#), [211](#), [213–215](#)
- printHelpMessage method, [235](#)
- printing, [30](#)
- println method, [60](#), [242](#), [364](#)
- printPretty method, [64](#)
- printRetAddr method, [208–211](#)
- printSomeEnvelopes method, [209–213](#)
- printStackTrace method, [216–217](#)
- printTriple method, [69](#)
- printWeight method, [177–178](#)
- private access
 - defined, [472](#)
 - working with, [173–175](#)
- products, [37](#)
- program counters
 - incrementing, [7](#)

- purpose of, [5](#)
- program files
 - for Macintosh installation, [404](#)
 - for Windows installation, [400–401](#)
- properties of objects, [118](#)
- protected access, [173](#)
 - defined, [472](#)
 - working with, [177–178](#)
- public access
 - defined, [472](#)
 - working with, [173–174](#)

Team LIB

4 PREVIOUS NEXT 5

Team LIB

← PREVIOUS

NEXT →

Index

Q

QTJava, [401](#)

question marks (?) in ternary operator, [ZZ](#)

quotients, [3Z](#)

Team LIB

← PREVIOUS

NEXT →

Index

R

- ␣ character, [264–265](#)
- radio buttons
 - defined, [472](#)
 - working with, [299–300](#), [300](#)
- RadioBoats class, [299–300](#)
- RAM, [248–249](#). See also [memory](#)
- random method, [243](#)
- RandomAreas class, [244](#)
- read method, [254](#)
- Read10Bytes class, [254–255](#)
- readBoolean method, [260](#)
- readByte method, [260](#)
- Reader class, [263](#)
- readers, [263](#), [264](#), [472](#)
- reading, [249](#)
 - bytes, [254–255](#), [256](#)
 - data, [259–261](#), [259](#), [262](#)
- readLine method, [265–266](#)
- readUTF method, [260](#), [263](#)
- ReadWithChain class, [260](#)
- reconfigure method, [377–378](#), [386–387](#)
- Rectangle class, [133](#)
- rectangle method, [281](#)
- rectangles
 - drawing, [279](#), [279](#)
 - filled, [281](#), [281](#)
- red color, [273–274](#)
- redrawing, [276](#)
- reference-dot notation, [126](#)
- reference variables, [109](#)
- references
 - to arrays, [110–112](#), [112](#)
 - with compiler, [155](#)
 - defined, [472](#)
 - equality of, [234](#), [234](#), [472](#)
 - to methods, [112–114](#), [113](#)
 - to objects, [121](#), [121](#), [190–192](#)
 - passing arguments by, [67](#)
 - for variables, [134–136](#)
- regions in layout managers, [317–319](#), [318–319](#)
- registers, [5](#), [6](#)
- RemoteEnvelopeCountException class, [219](#)
- removeActionListener method, [335](#)
- removeItemListener method, [352](#)
- repaint method
 - for ColorTest, [374](#)
 - for main display area, [377–378](#)
 - in Nim game, [348](#)
- result types in operations, [52–54](#), [53](#)
- return character, [264–265](#)
- return statement, [64](#)
- return types and values
 - defined, [472](#)
 - for methods, [60–61](#), [64–65](#)
- ReusesNames class, [69](#)
- RIGHT area, [315](#)
- right-shift operation, [42–44](#), [43](#)
- row major order
 - defined, [472](#)

in text areas, [310](#)
rows in text areas, [310](#)
Run Lightspeed option, [10](#)
runtime exceptions, [205–208](#)
RuntimeException class, [205](#)

Team LIB

◀ PREVIOUS

NEXT ▶

Index

S

- safety net catch blocks, [213](#)
- Sans Serif fonts, [285–286](#)
- scalability
 - defined, [473](#)
 - of events, [330](#)
- scientific notation
 - defined, [473](#)
 - purpose of, [24](#)
- scope
 - defined, [473](#)
 - in loops, [97](#)
 - of methods, [68–69](#)
- scripts, [398](#)
- Scrollbar class, [312](#)
- scrollbars
 - creating, [312–313](#), [313](#)
 - events from, [355–356](#), [356](#)
 - in text areas, [310–312](#), [311–312](#)
- SDK (Software Development Kit), [396–397](#)
- semicolons (;)
 - in classpath elements, [169](#)
 - in declarations, [27](#)
 - in do-while loops, [86](#)
 - in for loops, [87](#), [91](#)
- Serif fonts, [285–286](#)
- serifs, [473](#)
- setBackground method
 - in Button, [294](#)
 - for frames, [271](#), [275–276](#)
- setBounds method, [325](#)
- setColor method, [277](#)
- setColorScheme method, [183–184](#)
- setEnabled method, [349](#)
- setFont method, [284–285](#), [294](#)
- setForeground method, [294](#)
- setLayout method, [313–314](#), [325](#)
- setLocation method, [325](#)
- setNumEnvelopesInStock method, [210–211](#)
- setSalary method, [175](#)
- setSize method, [272](#), [282](#), [325](#), [367](#)
- setters
 - with data hiding, [173](#)
 - defined, [471](#)
- setTitle method, [271](#)
- setValues method, [183–184](#)
- setVisible method
 - for dialog boxes, [367](#)
 - for frames, [272–273](#), [289](#), [333](#)
- SeveralObjectsLab animated illustration, [123–125](#), [125](#)
- shifting operations
 - defined, [473](#)
 - process, [42–44](#), [42–44](#)
- ShiftLab animated illustration, [43–44](#), [44](#)
- short-circuit operators
 - in Boolean operations, [49–50](#)
 - defined, [473](#)
- short data type
 - range of, [21](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- ShowButton class, [293](#)

- ShowMeATrace class, [207](#)
- side effects
 - defined, [473](#)
 - with methods, [64–65](#)
- signed integer types
 - defined, [473](#)
 - summary, [21](#)
- SimCom computer, [5–8](#), [6](#)
 - benefits of, [11–12](#)
 - working with, [8–11](#), [9](#)
- Simple Base 2 animated illustration, [21–24](#), [22–23](#)
- Simple Event Lab animated illustration, [336–339](#), [336–339](#)
- Simple Exception Lab animated illustration, [203](#), [204](#)
- Simple Input Lab animated illustration, [255](#), [256](#)
- Simple Output Lab animated illustration, [252](#), [253](#)
- SimpleActionListener class, [334](#)
- SimpleChoice class, [302](#)
- SimpleFlow class, [314–315](#)
- SimpleNim class, [342–351](#), [342–344](#), [346](#), [349](#)
- sin method, [243](#)
- single-line comments, [36](#)
- size
 - of arrays, [103–105](#)
 - of canvas, [380](#)
 - of dialog boxes, [367](#)
 - of fonts, [284](#)
 - of frames, [272](#)
 - with layout managers, [314](#), [317](#)
 - of strings, [233](#)
- slashes (/)
 - for comments, [36](#)
 - in compound assignment, [51](#)
 - for division, [37](#)
- Software Development Kit (SDK), [396–397](#)
- source code, [16](#)
 - creating
 - in Macintosh, [404](#)
 - in Windows, [400–401](#)
 - defined, [473](#)
- South region, [317–319](#)
- specifying files in final project, [365–368](#), [366](#)
- SpringLayout layout manager, [325](#)
- square brackets ([]) for arrays, [102](#)
- squares, [280](#)
- stack traces
 - checked exceptions with, [216–217](#)
 - defined, [473](#)
 - runtime exceptions with, [206–208](#)
- startsWith method, [233](#), [385](#)
- statements
 - declarations, [27](#)
 - in for loops, [91](#)
- static classes, [473](#)
- static data, [128–130](#)
- static methods, [126–127](#), [130–132](#)
- static modifier, [60](#)
- Step Lightspeed option, [10](#)
- STORE opcode, [7](#)
- storeCubes method, [207–208](#)
- storeOneCube method, [207–208](#)
- String class, [188](#)
 - API pages for, [226](#)
 - for command-line arguments, [235–236](#), [235](#)
 - working with, [229–234](#), [231–232](#), [234](#)
- string concatenation, [233](#), [237–238](#)
 - defined, [473](#)
 - lab for, [238–239](#), [238–239](#)
- StringLab animated illustration, [231–232](#), [231–232](#)

- styles of fonts, [284–285](#)
- SUB opcode, [7](#)
- subclasses
 - defined, [473](#)
 - inheritance by, [140–142](#)
 - protected access with, [177](#)
- Submarine class, [149–150](#), [149](#)
- subpackages, [165](#)
- substring method, [233](#), [382](#), [385](#)
- subtraction
 - basic operator for, [37](#)
 - decrement operator for, [45–46](#)
- subtractive primary colors, [273](#), [473](#)
- sums, [37](#)
- super keyword, [151–152](#)
- supercategories, [142](#)
- superclasses, [140–142](#)
 - defined, [473](#)
 - inheritance from, [142–145](#), [144–145](#)
- Swing toolkit
 - for GUI, [271](#)
 - for layout managers, [325](#)
- switch statements, [77–79](#)
 - break statements in, [79–81](#)
 - default statements in, [79–80](#)
- System class, [241–243](#)
- System.exit call, [333](#), [343](#)

Index

T

- TAlnnaFrame class, [311](#)
- Talker interface, [188](#)
- tan method, [243](#)
- ternary operator
 - defined, [473](#)
 - operation of, [76–77](#)
- text
 - drawing, [283–286](#), [284–286](#)
 - in final project, [379](#), [379](#)
- text areas
 - events from, [353–355](#), [354–355](#)
 - working with, [310–312](#), [311–312](#)
- text characters, [25](#)
- text fields
 - events from, [353–355](#), [354–355](#)
 - in flow layout managers, [316](#)
 - working with, [309–310](#), [310](#)
- text files, [263](#)
- text listeners, [353](#)
- TextAreaNim class, [344–345](#)
- TextField constructor, [309](#)
- TextListener interface, [353](#)
- textValueChanged method, [353](#)
- TFs class, [309–310](#)
- Thermometer class, [135](#)
- this keyword, [131](#)
- this-reference notation, [131](#)
- threads
 - defined, [473](#)
 - in event-driven programs, [332–333](#)
- ThreeOvals class, [280](#)
- throw keyword, [200](#), [473](#)
- throwing exceptions
 - checked exceptions, [217–220](#)
 - process, [200–201](#)
- throws keyword, [200](#)
- tildes (~) in bitwise operations, [40](#), [40](#)
- toLowerCase method, [230–232](#), [231–232](#)
- toString method, [226](#)
 - in Object, [236–238](#)
 - in String, [239](#)
- toThe5th method, [59–61](#)
- toUpperCase method, [230–232](#), [231–232](#)
- traditional comments, [36](#)
- Transport class, [149–150](#), [149](#)
- Triangle class, [133](#)
- trim method, [233](#)
- trinary operator, [37](#)
- true value, [25](#)
- truncation
 - defined, [473](#)
 - with division, [40](#)
- try blocks, [205](#)
 - defined, [473](#)
 - working with, [202–203](#)
- Tuna class, [177–178](#)
- two-dimensional arrays, [106–107](#), [106](#)
- TwoBars class, [312–313](#)
- two's complement format

defined, [473](#)
for integer types, [21-23](#), [22](#)

Team LIB

← PREVIOUS

NEXT →

Index

U

- unary operators, [37](#)
 - arithmetic, [44–46](#)
 - defined, [473](#)
- Unicode standard, [25](#)
 - defined, [474](#)
 - for files, [257](#), [263](#)
- unnamed packages, [176](#)
- updates
 - defined, [474](#)
 - in for loops, [87](#), [87](#)
- upper case characters, [230–232](#), [231–232](#)
- useColor method, [183–184](#)
- UsesListener class, [334–335](#)
- UsesMethods class, [59](#)
- UTF standard
 - defined, [474](#)
 - for files, [257](#), [263](#)

Index

V

- values
 - vs. addresses, [12](#), [110](#)
 - logical, [25](#)
 - in memory, [110](#)
 - passing arguments by, [67–68](#)
- variable-width fonts
 - defined, [474](#)
 - vs. monospaced, [285](#)
- variables, [26](#)
 - for array size, [103](#)
 - for color, [274](#)
 - declaring, [27](#)
 - final, [180](#)
 - in for loops, [97](#)
 - inheritance with, [143](#)
 - instance, [130](#), [470](#)
 - for objects, [121–122](#)
 - polymorphism with, [155–156](#)
 - references for, [134–136](#)
 - scope of, [68–69](#)
- verbose option in java, [236](#)
- version option in java, [403](#)
- vertical bars (|)
 - for bitwise operator, [40–41](#)
 - for boolean operator, [46–48](#)
 - in compound assignment, [51](#)
 - for short-circuit operator, [49](#)
- vertical scrollbars, [312–313](#)
- VerySimple class, [28–29](#), [35](#)
- VerySimple2 class, [29–30](#)
- virtual computers, [19](#)
 - defined, [474](#)
 - JVM. See [JVM \(Java Virtual Machine\)](#)
 - SimCom, [12](#)
- visibility
 - of dialog boxes, [367](#)
 - of frames, [272–273](#)
- vocalHypotSquared method, [65](#)
- voltages, [2](#)

Index

W

- WaterTransport class, [149–150](#), [149](#)
- WeatherStation class, [135](#)
- West region, [317–319](#), [318–319](#)
- while loops, [82–86](#), [84–85](#)
- WhileLab animated illustration, [83–86](#), [84–85](#)
- white space
 - defined, [474](#)
 - in program code, [34–36](#)
- width
 - of canvas, [380](#)
 - of dialog boxes, [367](#)
 - with result types, [52–54](#), [53](#)
 - in text areas, [310](#)
- Windows computers, downloading and installing Java on, [396–401](#), [397](#)
- Worker class, [140–141](#)
 - constructors for, [146–147](#)
 - as subclass, [143](#), [143](#), [145–146](#)
- wrapper classes, [240](#)
 - benefits of, [241](#)
 - defined, [474](#)
- write method, [251](#)
- Write10Bytes class, [251–252](#)
- writeByte method, [257](#)
- writeChar method, [257](#)
- Writer class, [263](#)
- writers, [263](#), [264](#), [474](#)
- writeShort method, [257](#)
- writeUTF method, [257](#), [263](#)
- WriteWithChain class, [258](#)
- writing, [249](#)
 - bytes, [251–252](#), [253](#)
 - data, [256–259](#), [257](#)

Team LIB

← PREVIOUS

NEXT →

Index

X

X39 class, [241](#)

X39RevB class, [242–243](#)

Team LIB

← PREVIOUS

NEXT →

Team LIB

← PREVIOUS

NEXT →

Index

Z

Zebra class, [180–182](#)

zeroth array components, [104](#)

zip files, [399](#)

Team LIB

← PREVIOUS

NEXT →

List of Figures

Chapter 1: An Introduction to Computers That Will Actually Help You in Life

[Figure 1.1](#): A bit

[Figure 1.2](#): A byte

[Figure 1.3](#): Several bytes

[Figure 1.4](#): SimCom architecture

[Figure 1.5](#): Opcode and argument bits

[Figure 1.6](#): SimCom in action

Chapter 2: Data

[Figure 2.1](#): Assembly language

[Figure 2.2](#): Compiled language

[Figure 2.3](#): Evolution of a Java application

[Figure 2.4](#): SimpleBase2Lab

[Figure 2.5](#): A base-2 odometer

[Figure 2.6](#): An example of two's complement

[Figure 2.7](#): Two's complement lab

Chapter 3: Operations

[Figure 3.1](#): EvaluatorLab

[Figure 3.2](#): EvaluatorLab after evaluation

[Figure 3.3](#): The unary bitwise operator ~

[Figure 3.4](#): Bitwise "and"

[Figure 3.5](#): Left-shift: <<

[Figure 3.6](#): Bitwise right-shift: >>>

[Figure 3.7](#): Numeric right-shift: >>

[Figure 3.8](#): ShiftLab

[Figure 3.9](#): ShiftLab after shifting

[Figure 3.10](#): BoolLab: initial screen

[Figure 3.11](#): BoolLab after execution

[Figure 3.12](#): Data type width, not to scale

[Figure 3.13](#): Data type width relationships

Chapter 4: Methods

[Figure 4.1](#): MethodLab

[Figure 4.2](#): MethodLab after animating

[Figure 4.3](#): Numeric type widths

Chapter 5: Conditionals and Loops

[Figure 5.1](#): While Lab: initial display

[Figure 5.2](#): While Lab with modified test expression

[Figure 5.3](#): While Lab after execution

[Figure 5.4](#): A common loop usage

[Figure 5.5](#): A cycloid

[Figure 5.6:](#) NestedLoopLab: initial display

[Figure 5.7:](#) NestedLoopLab: 8:15

[Figure 5.8:](#) NestedLoopLab with a loop

[Figure 5.9:](#) NestedLoopLab with nested loops

Chapter 6: Arrays

[Figure 6.1:](#) A new array

[Figure 6.2:](#) A used array

[Figure 6.3:](#) A two-dimensional array

[Figure 6.4:](#) BoolArrayLab

[Figure 6.5:](#) BoolArrayLab drawing a parabola

[Figure 6.6:](#) Accessible and inaccessible memory

[Figure 6.7:](#) An array of bytes in inaccessible memory

[Figure 6.8:](#) Reference and array

[Figure 6.9:](#) Two references, one array

[Figure 6.10:](#) CreateArrayLab

Chapter 7: Introduction to Objects

[Figure 7.1:](#) Class as mental category

[Figure 7.2:](#) Reference and object

[Figure 7.3:](#) DataLab

[Figure 7.4:](#) Multiple objects

[Figure 7.5:](#) SeveralObjectsLab

[Figure 7.6:](#) SeveralObjectsLab reconfigured

[Figure 7.7:](#) SeveralObjectsLab reconfigured and executed

[Figure 7.8:](#) ObjectMethodLab

[Figure 7.9:](#) ObjectLifeCycleLab

[Figure 7.10:](#) ObjectLifeCycleLab after running a while

Chapter 8: Inheritance

[Figure 8.1:](#) A Simple inheritance hierarchy

[Figure 8.2:](#) Inherit Lab

[Figure 8.3:](#) Inherit Lab's class-editing dialog box

[Figure 8.4:](#) Object layers

[Figure 8.5:](#) Inheritance of `Officer`

Chapter 9: Packages and Access

[Figure 9.1:](#) Example package/ directory structure

[Figure 9.2:](#) Package as namespace

[Figure 9.3:](#) Initial directory structure

[Figure 9.4:](#) After compilation

[Figure 9.5:](#) After more compilation

[Figure 9.6:](#) Polymorphism revisited

[Figure 9.7:](#) Chart class and subclasses

Chapter 10: Interfaces

[Figure 10.1](#): Animal kingdom class inheritance

Chapter 11: Exceptions

[Figure 11.1](#): Simple Exception Lab

[Figure 11.2](#): Simple Exception Lab: final state with normal execution

[Figure 11.3](#): Advanced Exception Lab

[Figure 11.4](#): Choosing an exception type in Advanced Exception Lab

[Figure 11.5](#): Advanced Exception Lab reconfigured

Chapter 12: The Core Java Packages and Classes

[Figure 12.1](#): Structure of the API index

[Figure 12.2](#): Structure of the classes frame

[Figure 12.3](#): Class description

[Figure 12.4](#): Field/constructor/ method summaries

[Figure 12.5](#): StringLab

[Figure 12.6](#): StringLab: uppercase, 2 references

[Figure 12.7](#): StringLab: lowercase, 1 reference

[Figure 12.8](#): String references and objects

[Figure 12.9](#): Command-line arguments

[Figure 12.10](#): ConcatLab

[Figure 12.11](#): ConcatLab's Point3D class

[Figure 12.12](#): ConcatLab's Point3D class

Chapter 13: File Input and Output

[Figure 13.1](#): Simple Output Lab

[Figure 13.2](#): Simple Output Lab in progress

[Figure 13.3](#): Simple Input Lab in progress

[Figure 13.4](#): Output chaining

[Figure 13.5](#): Input chaining

[Figure 13.6](#): Data Chain Lab

[Figure 13.7](#): Data Chain Lab in progress: Text, writers, and readers

[Figure 13.8](#): Readers and writers

[Figure 13.9](#): Line number reader and file reader

Chapter 14: Painting

[Figure 14.1](#): A frame with boring contents

[Figure 14.2](#): Color Lab

[Figure 14.3](#): Color Lab with a predefined color

[Figure 14.4](#): Pixel coordinates

[Figure 14.5](#): A black line on a white background

[Figure 14.6](#): A rectangle

[Figure 14.7](#): Ovals and bounding boxes

[Figure 14.8](#): Three ovals

[Figure 14.9](#): Filled rectangle and ovals

[Figure 14.10](#): Original CenteredOval

[Figure 14.11](#): Resized CenteredOval

[Figure 14.12](#): The baseline

[Figure 14.13](#): Text and baseline in a frame

[Figure 14.14](#): Text in a frame

[Figure 14.15](#): Font Lab

[Figure 14.16](#): Font Lab with an exotic font

[Figure 14.17](#): Initial Frame Lab display

[Figure 14.18](#): Frame Lab with custom configuration

[Figure 14.19](#): The result of Figure 14.18

Chapter 15: Components

[Figure 15.1](#): A component sampler

[Figure 15.2](#): A button in a frame

[Figure 15.3](#): A fancy button

[Figure 15.4](#): A simple checkbox

[Figure 15.5](#): A checked checkbox

[Figure 15.6](#): Three checkboxes and a button

[Figure 15.7](#): Checkboxes as radio buttons

[Figure 15.8](#): Multiple checkbox groups

[Figure 15.9](#): A choice

[Figure 15.10](#): An expanded choice

[Figure 15.11](#): Two choices

[Figure 15.12](#): Choices with labels

[Figure 15.13](#): A menu in a menu bar

[Figure 15.14](#): A menu with a separator

[Figure 15.15](#): Hierarchical menus

[Figure 15.16](#): Two text fields

[Figure 15.17](#): A text area

[Figure 15.18](#): Multiple checkbox groups

[Figure 15.19](#): A text area with scroll bars

[Figure 15.20](#): A pair of disappointing scrollbars

[Figure 15.21](#): Flow layout manager

[Figure 15.22](#): Wider

[Figure 15.23](#): Narrower

[Figure 15.24](#): Left-aligned

[Figure 15.25](#): Flow Lab

[Figure 15.26](#): Scrollbar at North

[Figure 15.27](#): North and South occupied

[Figure 15.28](#): North, East, and West occupied

[Figure 15.29](#): North, East, West, and Center occupied

[Figure 15.30](#): A panel in a frame

[Figure 15.31](#): Layout lab

[Figure 15.32](#): Layout lab's frame editing dialog

[Figure 15.33](#): Layout Lab with an added panel

[Figure 15.34](#): A button in a panel in a frame

[Figure 15.35](#): Layout Lab makes it so

[Figure 15.36](#): No layout manager

Chapter 16: Events

- [Figure 16.1](#): A GUI waiting for events
- [Figure 16.2](#): A button that sends events
- [Figure 16.3](#): Simple Event Lab: initial screen
- [Figure 16.4](#): Simple Event Lab with simulated buttons
- [Figure 16.5](#): Simple Event Lab with a listener class
- [Figure 16.6](#): Simple Event Lab with a listener object
- [Figure 16.7](#): Simple Event Lab continued
- [Figure 16.8](#): One listener object for many buttons
- [Figure 16.9](#): Simple Nim GUI
- [Figure 16.10](#): Nim Lab
- [Figure 16.11](#): Nim, with output to a text area
- [Figure 16.12](#): Nim with graphical output
- [Figure 16.13](#): Nim with graphical output, game in progress
- [Figure 16.14](#): Enabled and disabled buttons
- [Figure 16.15](#): Nim with disabled buttons
- [Figure 16.16](#): Check box and choice
- [Figure 16.17](#): Receiving events from a check box and a choice
- [Figure 16.18](#): Event Lab
- [Figure 16.19](#): Scrollbar and text field

Chapter 17: Final Project

- [Figure 17.1](#): Final Project
- [Figure 17.2](#): Final Project, with lines
- [Figure 17.3](#): Menu schematic
- [Figure 17.4](#): Teting the menu's look
- [Figure 17.5](#): Window, Frame, and FileDialog
- [Figure 17.6](#): File dialog box configured for opening
- [Figure 17.7](#): Too many radio buttons
- [Figure 17.8](#): Testing color selection
- [Figure 17.9](#): GUI layout
- [Figure 17.10](#): Positioning text

Appendix A: Downloading and Installing Java

- [Figure A.1](#): Windows SDK file layout

List of Tables

Chapter 1: An Introduction to Computers That Will Actually Help You in Life

[Table 1.1:](#) Opcodes

Chapter 2: Data

[Table 2.1:](#) Java's Integer Data Types

[Table 2.2:](#) Java's Floating-Point Data Types

[Table 2.3:](#) Java's Primitive Data Types

[Table 2.3:](#) Naming Consistency

Chapter 3: Operations

[Table 3.1:](#) Binary Bitwise Operations

[Table 3.2:](#) Comparison Operators

[Table 3.3:](#) Compound Assignment

[Table 3.4:](#) Ranges of Numeric Types

[Table 3.5:](#) Binary Arithmetic Result Types

[Table 3.6:](#) Operator Precedence

Chapter 8: Inheritance

[Table 8.1:](#) References, Variables, and Methods

Chapter 9: Packages and Access

[Table 9.1:](#) Legal Access Modes for Overriding Methods

Chapter 12: The Core Java Packages and Classes

[Table 12.1:](#) String Concatenation Conversion Rules

[Table 12.2:](#) Wrapper Class Names

Chapter 13: File Input and Output

[Table 13.1:](#) Byte -1 vs. Int -1

Chapter 14: Painting

[Table 14.1:](#) Combining Additive Primary Colors

Back Cover

This is the first effective Java book for true beginners. Sure, books before now focused on basic concepts and key techniques, and some even provided working examples on CD. Still, they lacked the power to transform someone with no programming experience into someone who sees, who really “gets it.”

Working with *Ground-Up Java*, you will definitely get it. This is due to the clarity of Phil Heller’s explanations, and the smoothly flowing organization of his instruction. He’s one of the best Java trainers around.

But what’s really revolutionary are his more than 30 animated illustrations. Each of these small programs, visual and interactive in nature, vividly demonstrates how its source code works. You can modify it in different ways, distinctly altering the behavior of the program. As you experiment with these tools—and you can play with them for hours—you’ll gain both the skills and the fundamental understanding needed to complete each chapter’s exercises, which steadily increase in sophistication. No other beginning Java book can take you so far, so quickly, and none will be half as much fun.

About the Author

Philip Heller is a consultant, author, educator, and novelist. He is the lead author for Sybex’s best selling *Java Certification Study Guide* and *Java Exam Notes* as well as a leading educator for Java University and a well-known speaker on Java topics. Phil helped create the Java programmer and developer exams for Sun and is their leading certification trainer. Phil is currently writing the second volume in the *Grandfather Dragon* series.

Ground-Up Java

Philip Heller

Associate Publisher: Joel Fugazzotto
Acquisitions Editor: Denise Santoro Lincoln, Tom Cirtin
Developmental Editor: Tom Cirtin
Production Editor: Dennis Fitzgerald
Technical Editor: Marcus Cuda
Copyeditor: Sean Medlock
Composer: Maureen Forsys, Happenstance Type-O-Rama
Graphic Illustrator: Jeffrey Wilson, Happenstance Type-O-Rama
CD Coordinator: Dan Mummert
CD Technician: Kevin Ly
Proofreaders: Emily Husan, Laurie O'Connell, Nancy Riddiough
Indexer: Ted Laux
Cover Designer/Illustrator: Richard Miller, Calyx Deisgns

Copyright © 2004 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 2003110719

ISBN: 0-7821-4190-0

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the United States and/or other countries.

Screen reproductions produced with FullShot 99. FullShot 99 © 1991–1999 Inbit Incorporated. All rights reserved. FullShot is a trademark of Inbit Incorporated.

The CD interface was created using Macromedia Director, COPYRIGHT 1994, 1997-1999 Macromedia Inc. For more information on Macromedia and Macromedia Director, visit <http://www.macromedia.com>.

TRADEMARKS: SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

*To Laura, on whose violin
Are played the songs of spheres and heroes,
Above this world's mortal din,
Above the plane of ones and zeroes.*

Acknowledgements

First and foremost gratitude to Denise Santoro Lincoln, Tom Cirtin, and Steve Cavin. Thanks to Michelle, Ricardo, and everyone at PB&G Productions for keeping me out dancing when I should have been writing. Thanks always to Simon Roberts, Suzanne Blackstock, and Kathy Collina. And thanks to all the aces at Sybex: Dennis Fitzgerald, Sean Medlock, Kevin Ly, Dan Mummert, and Maureen Forsys and Jeff Wilson at Happenstance Type-O-Rama.

Introduction

Overview

This book is unique. There's nothing like it. It is the first of its kind. It's important that you understand why, so please read on.

For a long time I thought it was impossible to write an introductory Java programming book that could be understood by people with no programming experience. It would be like a fish writing about water. No one has better knowledge of the subject matter, but it takes more than that to introduce a topic to a newcomer. Fish are intimately accustomed to water, and they can't relate to us land mammals, who need to have everything explained and broken down. A fish might say, "Wiggle your tail fin to swim forward, and don't forget to use your gills." That would be glaringly obvious to another fish, but useless to you and me. It's *hard* for a fish to imagine what life would be like without tail fins or gills. A book about water, even if the wisest fish in the ocean wrote it, would be full of accurate, but useless, information.

The same is true about Java. Programming is a craft, like playing a musical instrument or glassblowing. And like any other craft, it has its conventions, jargon, and techniques. For practitioners of the craft, those conventions, jargon, and techniques become deeply ingrained habits, household language, and the events of everyday life. It's very difficult to write about one's own "habitat."

In the 1970's, a language called C became popular. In the 1980's, C was modified to support object-oriented programming. The modified language was called C++. This is an example of craft jargon. In C, the symbol "+" means, very broadly speaking, "a bit more." So C++ means "C and a bit more," and the meaning is clear to any C programmer.

The 1990's saw another evolution. C++ is a highly effective language, but it can also be difficult. Moreover, it had no innate support for recently invented technologies, such as high-resolution multi-color displays, databases, or the World Wide Web. The new evolution was called Java. The name isn't a play on words and it isn't an abbreviation for anything. Java abandoned the parts of C++ that had proved to be more trouble than they were worth, and it added support for modern technologies. Sometimes people called it "C++-++". There's another symbol, "-", that roughly means "a bit less." So "C++-++" means "C++ and a bit less and then a bit more."

Java caught on like a midsummer bonfire. A huge portion of the C and C++ programming population switched at once to Java and never looked back. Why were so many programmers able to make the switch so easily? I was one of them. I had been earning a living programming in C++. I took a year off to write a novel about some dragons. I ran out of money before I finished the novel. Luckily, it was a month after Java was introduced. Within weeks I considered myself a competent Java programmer, and within months I was teaching it and writing about it. The credit goes not to me but to the designers of Java. If you know C and C++, Java is easy. It's like learning Portuguese if you already speak Spanish and Italian. Like everyone else who learned Java at that time, I had years of experience with the concepts, techniques, and jargon that was needed.

But what about people who don't have any programming experience?

When I was learning Java, there were two books on the subject. Today there are thousands. (I'm responsible for a few of them.) Not one of them, *except the one that you're holding right now*, does a good job of presenting programming concepts from the ground up. The others are accurate for the most part, but they aren't helpful.

So I had to ask myself: can I introduce Java from the ground up, concept by concept? Eventually I realized that I could only do it if I could use something more than words and pictures. Which brings me to why this book is unique. It is unique because ...

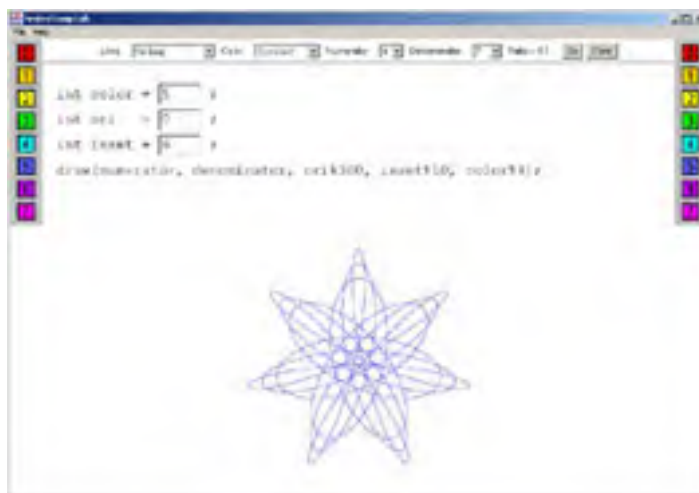
The Illustrations are Alive!

I realized that what I really wanted was a magic blackboard.

Think of a computer as a huge set of boxes, each box containing a number. The numbers represent text or colors or data, or whatever else can be modeled by a program. The numbers change over time in complicated ways. Describing the life cycle of a program is almost impossible if you can only use words and pictures. I wanted to create pictures that would change over time. And I wanted something beyond animated cartoons that would be the same each time you watched them. I wanted living illustrations that would respond to your curiosity. I wanted to give you the power to ask “what if ...” questions of the illustrations.

I wanted something that can only be done on a computer.

The CD-ROM that comes with this book has more than 30 *animated illustrations*. These are programs that you run on your computer. The book gives you complete instructions on how to use them. The illustration on the next page is an example.



This is a screenshot of NestedLoopLab, which appears in [Chapter 5, “Conditionals and Loops.”](#) The text in the upper-central part of the screen (“int color = 5” and so on) is Java code. The swirly image at the bottom is the result of running the code. The various controls let you vary the code, experimenting with different values until you get a feel for what the program is doing.

The animated illustrations are like training wheels on a bicycle. When you first learn to ride, there are so many things that can go wrong. Without training wheels you spend a lot of just time crashing and getting back up. Training wheels let you develop the right sense of balance. The animated illustrations won’t let you create code that crashes. They provide a safe environment in which you can develop the right sense of balance.

Later, of course, it’s time to take off the training wheels. At the end of each chapter you’ll find a set of exercises that will have you writing your own code. Suggested solutions to the exercises appear at the back of the book.

To the best of my knowledge, [Ground-Up Java](#) is the first book ever to use animated illustrations. So we have no data on how effective they are as a teaching tool. My guess is that they are worth their weight in gold. Everyone who has seen them has been very enthusiastic. But you are the most qualified judge. Try them! Please let me know what you think. You can e-mail your comments to groundupjava@sqsware.com. I’m especially interested in knowing which animated illustrations worked the best for you, and which ones didn’t. I’d also like to hear any suggestions you might have for more animations to appear in future revisions of this book. You are invited to be part of the development of animated illustrations as a new technology for learning.

And now...

Team LIB

← PREVIOUS

NEXT →

It's Time To Download and Install Java

Before you can start writing or running Java programs, you need to download some software. (The animated illustrations are Java programs, so they won't run if you don't do the download.)

Downloading is free. After Java is loaded on your hard drive, you have to follow a few steps to install it. These aren't difficult, but there's room for error, so please be careful. Complete instructions are explained in [Appendix A, "Downloading and Installing Java."](#)

And now you're ready. Have fun!

Team LIB

← PREVIOUS

NEXT →

Chapter 1: An Introduction to Computers That Will Actually Help You in Life

Overview

Java is a programming language that tells computers what to do. This chapter will look at what computers really are, what they can do, and how we use programming languages to control them.

We will begin by exploding the common myth that computers deal only with 0s and 1s. Once we establish what computers really process, we will look at the kind of processing they perform.

This is emphatically *not* an intellectual exercise. Spending a bit of effort here will make your life much easier in the chapters that follow. Many concepts that appear later in this book, such as data typing, referencing, and virtual machines, will make very little sense unless you understand the underlying structure of computers. Without this understanding, learning to program can be confusing and overwhelming. With the right fundamentals, though, it can be enjoyable and stimulating.

Memory: Not Exactly 0s and 1s

No doubt you've heard that computers only process 0s and 1s. This can't possibly be true. Computers are used to count votes in elections, so they must be capable of counting past 1. Computers are also used to model the behavior of subatomic particles whose masses are tiny fractions, so they must be capable of processing fractions as well as whole numbers. They're used for writing documents, so they must be capable of processing text as well as numbers.

On the most fundamental level, computers do not process 0s and 1s, or whole numbers, or fractions, or text. Computers are electronic circuits, so all they really process is electricity. Computer components are designed so that their internal voltages are either approximately zero or approximately 5 or 6 volts. When part of a computer circuit carries a voltage of 5 or 6 volts, we say that it has a value of 1. When part of a circuit carries zero voltage, we say that it has a value of 0. (Fortunately, this is all the electronics knowledge you need to become a master programmer.)

It's all a matter of interpretation. Voltages are interpreted as 0s and 1s. As you'll see later in this chapter and in [Chapter 2, "Data,"](#) the 0s and 1s are organized into clusters that are interpreted as numbers. More sophisticated parts of the computer interpret those numbers as codes that represent fractions, or text, or colors, or images, or any of the other myriad classes of objects that can be represented in a computer.

A modern computer contains billions of microscopic components, each of which has a value of 0 or 1. Any circuit where we only care about the approximate values of the voltages is known as a *digital circuit*. Computers that are made of digital circuitry are known as *digital computers*.

Note The opposite of digital is *analog*. In an analog circuit, we care about the exact voltages of the components. Analog circuits are ideal for certain applications, such as radios and microwave ovens, but they don't work so well for computers. Analog computers were used in the 1940s, but they were an evolutionary dead end. All modern computers are digital.

One simple but useful type of digital circuit is known as *memory*. A memory circuit just stores a digital value (0 or 1, because we programmers don't have to think about voltages). A single unit of memory is called a *bit*, which is an abbreviation for "binary digit." You can think of a bit as a microscopic box, the contents of which are available to the rest of the computer. From time to time the computer might change the contents. Bits are usually drawn as shown in [Figure 1.1](#).



Figure 1.1: A bit

Bits are usually organized in groups of eight, known as *bytes*. [Figure 1.2](#) shows a byte that contains an arbitrary combination of 0s and 1s.

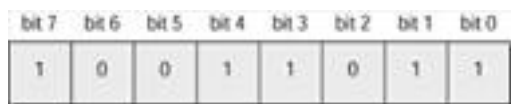


Figure 1.2: A byte

Note that the individual bits are numbered from right to left, and that the numbering starts from 0. Computer designers always start numbering things from 0 rather than 1. This is true whether they are numbering bits in a byte, bytes in memory (as we are about to see), or components in an array (as we will see in [Chapter 6](#)).

A byte can contain 256 combinations of bit values: 2 possibilities for bit #0 times 2 possibilities for bit #1 times 2 possibilities for bit #3, and so on up through bit #7.

If you looked at a computer through a microscope and saw the byte shown in [Figure 1.2](#), you might wonder what value it contained. You would see the 0s and 1s, but what would they mean? It's a great question that has no good answer. A byte might represent an integral number, a fraction, part of an integer or fraction, a character in a document, a color in a picture, or an instruction in a program. It all depends on the byte's context. As a programmer, you are the one who dictates how each byte will be interpreted.

Memory Organization

Typically, a modern personal computer contains several hundred million bytes of memory. The prefix *mega* (abbreviated *M*) means million, so we could also say that a computer has several hundred megabytes or *MB*. Programs and programmers need a way to distinguish one byte from another. This is done by assigning to each byte a unique number, known as the byte's *address*. Addresses begin at 0. [Figure 1.3](#) shows 4 bytes.

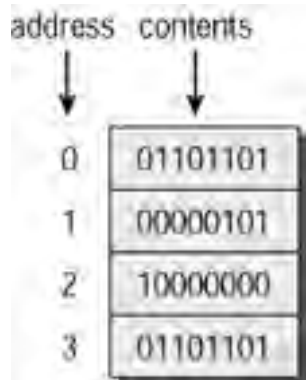


Figure 1.3: Several bytes

If [Figure 1.3](#) showed 512 MB and was drawn to the same scale, it would be about 2,000 miles high.

A single byte is not very versatile, because its value is limited to 256 possibilities. It doesn't matter whether the byte represents a number or a letter or anything else—in computer applications, 256 of *anything* isn't much of a range. For this reason, computers often use groups of bytes. Two bytes, taken together as a unit, can take on 256 times 256 possible values, or 65,536. Four bytes can take on 256 times 256 times 256 times 256 values, or 4,294,967,296. This is where it starts to be useful. Eight bytes can take on approximately 20 quintillion different values.

Memory is usually used in chunks of 1, 2, 4, or 8 bytes. (Later we will see that arrays and objects use chunks of arbitrary size.) The chunks can represent integral numbers, fractions, text, or any other kind of information. From this perspective, we can see that the statement "Computers only deal with 0s and 1s" is true only in a very limited sense.

Think of it this way: A computer is a digital circuit, and we think of its components as having values that represent 0s or 1s. But if we look one level below the digital components, we see only electricity, not numbers. And if we look one level above the digital components, we see that the bits are organized into chunks of 1 or more bytes that represent many types of information.

In addition to various types of data, memory can also store the instructions that operate on data. In the [next section](#), we will look at a very simple computer and see how instructions and data interact.

A Very Simple Computer

This chapter will introduce a very simple computer called SimCom. SimCom is imaginary. Or, to use a more respectable term, it is *virtual*. Nobody has ever built a SimCom, but it is simulated in one of the animated illustrations on the CD-ROM.

The processors that power your own computer, the Pentiums, SPARCs, and so on, are not very different qualitatively from SimCom. *Quantitatively*, however, there is a huge difference: the real processors have vastly more instructions, speed, and memory. SimCom is as simple as a computer can be while still being a useful teaching tool.

The point of this section is not to make you a master SimCom programmer. The point is to use SimCom to introduce certain principles of programming. Later in this book, the same principles will be presented in the context of Java. These principles include

- High-level languages
- Loops
- Referencing
- Two's complement
- Virtual machines

In this section, you will see some typical processor elements that are quite low-level. Modern programming languages like Java deliberately isolate you from having to control these elements. However, it is extremely valuable to know that they exist and what they do on your behalf.

The architecture of SimCom is very simple. There is a bank of 32 bytes of memory; each byte can be used as an instruction or as data. There is one extra byte, called the *register*, which is used like scratch paper. Another component, called the *program counter*, keeps track of which instruction is about to be executed. [Figure 1.4](#) shows the architecture of SimCom.

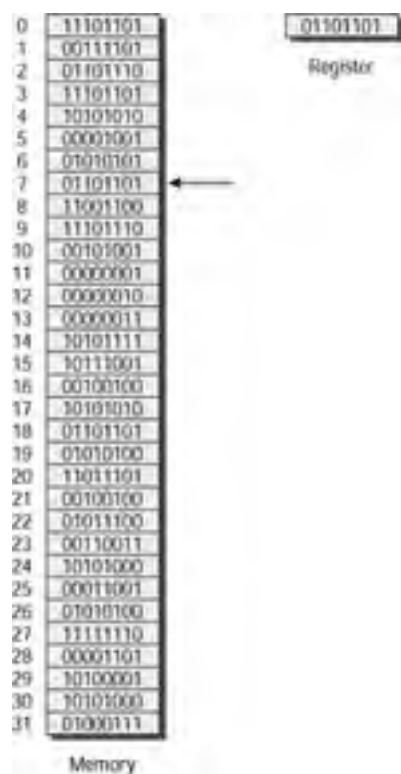


Figure 1.4: SimCom architecture

The arrow in the figure indicates the program counter. The next instruction to be executed will be byte #7. Note that byte addresses start at 0.

When SimCom starts up, it sets the program counter to 0. It then *executes* byte 0. (We'll see what this means in a moment.) Execution may change the register or a byte of memory, and it almost always changes the program counter. Then the whole process repeats: The instruction indicated by the program counter is executed, and the program counter is modified. This continues until SimCom is instructed to halt.

Bits 7, 6, and 5 of an instruction byte tell SimCom what to do. They are known as the *operation code* or *opcode* bits. Bits 4 through 0 contain additional instructions; they are called the *argument* bits. This division of bits is shown in [Figure 1.5](#).

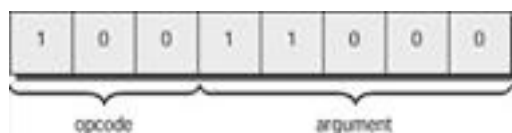


Figure 1.5: Opcode and argument bits

The SimCom computer has 7 opcodes. They are shown in [Table 1.1](#).

Table 1.1: Opcodes

Opcode	Function	Abbreviation
000	Load	LOAD
001	Store	STORE
010	Add	ADD
011	Subtract	SUB
100	Jump to different current instruction	JUMP
101	Jump if register is zero	JUMPZ
110 or 111	Halt	HALT

The 5 argument bits contain a value that is the base-2 address of a memory byte. The LOAD opcode copies the contents of this address into the register. For example, suppose the current instruction is 0000011. The opcode is 000 (LOAD), and the argument is 00011 (which is the base-2 notation for 3). When the instruction is executed, the value in byte #3 is copied into the register. Note that the value 3 is *not* copied into the register. The argument is never used directly; it is always an address whose contents are used.

The STORE opcode copies the contents of the register in the memory byte whose address appears in the argument. For example, 00100001 causes the register to be copied into byte #1.

The ADD opcode adds two values. One value is the value stored in the byte whose address appears in the argument. The other value is the contents of the register. The result of the addition is stored in the register. For example, suppose the register contains 00001100, and byte #1 contains 00000011. The instruction 01000001 causes the contents of byte #1 to be added to the contents of the register, with the result being stored back in the register. Note that the argument (00001) is used *indirectly*, as an address. The value 00001 is not added to the register; rather, 00001 is the address of the byte that gets added to the register.

The SUB opcode is like ADD, except that the value addressed by the argument is subtracted from the register. The result is stored in the register.

After each of these four opcodes is executed, the program counter is incremented by 1. Thus, control flows sequentially through memory. The remaining three opcodes alter this normal flow of control. The JUMP opcode does not change the register or memory; it just stores its argument in the program counter. For example, after executing 10000101, the next instruction to be executed will be the one at byte 00101, which is the base-2 notation for 5.

The JUMPZ opcode inspects the register. If the register contains 00000000, the program counter is set to the instruction's argument. Otherwise, the program counter is just *incremented* (that is, increased by 1) and control flows normally. This is a very powerful opcode, because it enables the computer to be sensitive to its data and to react differently to different conditions.

Finally, the HALT opcode causes the computer to stop processing.

Let's look at a short program:

```
00000100
01000100
00100100
11000000
```

The first thing to notice about this program is that it's hard to read. Let's translate it to a friendlier format:

```
LOAD    4
ADD     4
STORE   4
HALT
```

The program doubles the value in byte #4. It does this by copying the value into the register, then adding the same value into the register, and then storing the result back in byte #4.

This example shows that anything is better than programming by manipulating 0s and 1s. These spelled-out opcodes and base-10 numbers are a compromise between the binary language of computers and the highly structured and nuanced language of humans. The LOAD 4 notation is known as *assembly language*. In assembly language, a line of code typically corresponds to a single computer instruction, and the programmer must always be aware of the computer's architecture and state. An *assembler* is a program that translates assembly language into binary notation.

Playing with SimCom

Unfortunately we couldn't package a SimCom with every copy of this book, but we have done the next best thing. The first animated illustration on the book's CD is a simulation of a SimCom in action.

Note If you don't already have Java installed on your computer, now is the time. If you're not sure how, please refer to [Appendix A, "Downloading and Installing Java."](#) which walks you through the entire process. Throughout this book you

will be invited to run an animated illustration program, and you will be given a command to type into your machine. It will all make sense after you go through [Appendix A](#).

To run the SimCom simulation, type the following at your command prompt:

```
java simcom.SimComFrame
```

The simulation allows you to load and run preexisting programs or create your own programs. [Figure 1.6](#) shows the simulation in action.

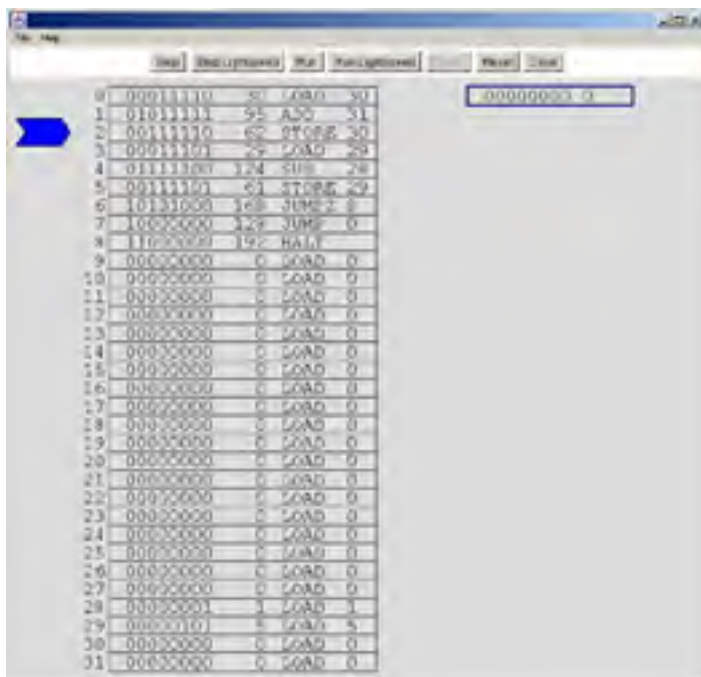


Figure 1.6: SimCom in action

Each byte of memory is displayed in three formats: base-2, base-10, and opcode-plus-argument. The register is only displayed in base-2 and base-10; since the register is never executed, there is no value in displaying which instruction *would* be executed. You can change any byte in memory by first clicking inside that byte. This will highlight and select the byte for editing. Then, if you click on the base-10 region, you will get a panel that lets you select a new base-10 value. If you click on the opcode region, you will get a panel that lets you select a new opcode. To change the argument, first click on the argument region of the selected byte. As you move the mouse, the closest byte address will light up. When the address you want is highlighted, click on it to set it as the argument.

Try executing a very simple program. Click File, Scenarios in the File menu, and select Load/Add/Store/. This program adds bytes 10 and 11 (not the numbers 10 and 11, but the contents of the memory bytes whose addresses are 10 and 11), and stores the result in byte 12. Initially, bytes 10 and 11 both contain zero, so to see interesting results you will have to change their values. To see the program in action, click the Step button. This executes the current instruction in slow motion. To run continuously, click the Run button, which plays the animation until a HALT instruction is executed. If you get tired of the slow motion, you can click Step Lightspeed or Run Lightspeed to get instant results. The Reset button reinitializes memory and sets the program counter to zero.

Try storing relatively large values in bytes 10 and 11. The largest value a byte can store is 255. What happens if you try to add 5 + 255?

Change the program so that byte 11 is subtracted from byte 10. What happens if byte 10 contains 5 and byte 11 contains 6?

When you are ready for a more interesting program, click Scenarios, Times 5 in the File menu. This program multiplies the contents of byte 31 by 5 and stores the result in byte 30. Experiment with a few values in byte 31 to convince yourself that it works. Remember to click the Reset button after each run.

This program might seem needlessly complicated. It's too bad the SimCom instruction set doesn't include a multiply opcode, but since it doesn't, wouldn't the following program be more straightforward?

```
LOAD    31
ADD     31
ADD     31
ADD     31
ADD     31
STORE  30
HALT
```

This is definitely more straightforward, but it is also less flexible than the version SimCom uses. That version uses a *loop*, a powerful construct that appears in all programming languages. Note that initially, byte 29 contains 5; this byte is a *loop counter* that controls how many times the loop will be executed. Lines 0 through 3 add whatever is in byte 31 (the value to be quintupled) to whatever is in byte 30 (the accumulating result). Then lines 3 through 5 subtract 1 from the loop counter. If the loop counter reaches zero, line 6 causes a jump to a HALT instruction. If the decremented loop counter has not yet reached zero, line 7 causes a jump back to line 0, which is the beginning of the loop.

Reset the Times 5 program. Change the value in byte 29 (the loop counter) from 5 to 6. Put a reasonable value in byte 31 and run the program. Notice that the program now multiplies

by 6. This is to be expected, because the value in byte 31 has been added one extra time to the accumulated result.

Now you can see how the looping version is more flexible than the repeated-addition version shown earlier. To modify the looping version so that it multiplies by 10 instead of 5, you just have to change the loop counter in byte 29. In the repeated-addition version, you have to make sure you add the right number of ADD 31 lines, and then make sure the STORE 30 and HALT lines are intact. That may not seem unreasonable to you, but what if you want the program to multiply by 30? With the looping version, you just change the loop counter. With the repeated-addition version, you will run out of memory.

As you experiment with the SimCom simulation, you will probably notice a few things:

- Specifying an instruction by selecting an opcode and an argument is much easier than figuring out what the base-10 value should be.
- Even so, SimCom programming isn't very easy.
- When you look at any byte, you can't tell if it is supposed to be an instruction or a value. For example, a byte that contains 100 might mean one hundred, or it might mean SUB 4.

The first two points suggest the need for higher-level programming languages. Hopefully, such languages will support sophisticated operations like multiplication and looping.

The Lessons of SimCom

The point of presenting SimCom in this chapter was to expose you to certain basic functions of programming. Those were high-level languages, loops, referencing, two's complement, and virtual machines. Now that you've been exposed, we can look at how SimCom supports those functions.

Programming with opcodes and arguments is certainly easier than specifying base-10 or (worse yet) base-2 values. But SimCom still forces you to think on the microscopic level. In the Times5 program, you have to remember that byte 29 is the loop counter and byte 30 is the accumulated result. You always have to remember what's going on in the register. High-level languages like Java isolate you from the details of the computer you're programming. (That probably sounds like a good thing, now that you have suffered through SimCom.)

Loops are basic to all programming. Computers are designed to perform repetitive tasks on large data sets, such as printing a paycheck for each employee, displaying each character of a document, or rendering each pixel of a scanned photograph. Loops are difficult to create on SimCom, because everything is hard on SimCom. Java uses simple and powerful looping constructs.

We will cover referencing much later in this book, in [Chapter 6, "Arrays."](#) For now, you've had a preview. Remember how SimCom never directly operated with an instruction's argument? The argument was always used as the address of the value to be loaded, added, etc. Now you should be used to the difference between the *address* of a byte and the *value* in that byte. When you program in Java, you don't have to worry about the address of your data, but you still have to think about its location. This will make more sense later on. For now, it's enough to understand the distinction between the value of data and the location of data.

Two's complement is a convention for storing negative numbers. On its surface, SimCom seems to deal only with positive numbers (and zero, of course). But subtraction is supported, and subtraction can lead to negative numbers. If you did the exercise where you modified the LoadAddStore program to make it subtract, you noticed that SimCom thinks 5 minus 6 equals 255. In a way, this is actually correct.

SimCom does not really exist. When you run the animated illustration, there is no actual SimCom computer doing the processing. The program simulates the computer's activity. Thus, SimCom is an imaginary processor that produces real results. As stated earlier in this chapter, an imaginary computer that is simulated on a real one is known as a *virtual computer*. You might have heard of the *JVM*, or *Java Virtual Machine*. Java programs, like SimCom programs, run on a virtual computer.

There is a powerful benefit to this arrangement. When you buy software for your personal computer, you have to check the side of the box to make sure the product works on your platform. If you own a Windows PC, it is useless to buy Macintosh software, just as it is useless to buy SPARC software for a Mac. This is because different manufacturers use different kinds of processors. The binary opcode for addition on one processor type might mean subtract to another type, and might be meaningless to a third type. Thus, software vendors have needed to create a different product for each computer platform they want to support.

Virtual computers do not have this limitation. No matter what kind of computer you're using, SimCom loads when it executes 000, stores when it executes 001, and multiplies by 5 when it executes the Times5 program.

The Java Virtual Machine is much more complicated than SimCom, but the same principle applies. Any Java program will run the same on any hardware. Of course, the JVM itself varies from processor to processor. This is why you had to specify your platform when you downloaded Java. From the JVM's point of view, your platform is known as the *underlying hardware*.

Exercises

Note Every chapter in this book ends with exercises that test your understanding of the material and make you think about issues raised in later chapters. The solutions are in [Appendix B](#).

1. A cluster of eight bytes can take on approximately 20 quintillion different values. (One quintillion is a 1 followed by 18 zeroes, or 10^{18} .) Estimate the number of different values that a cluster of 16 bytes can have. Just estimate, do not count. Can you think of anything that comes in such quantities?
2. The SimCom animated illustration is written in Java. When you run the program, how many virtual machines are at work?
3. Write a SimCom program that adds 255 to the value in byte 31 and stores the result in byte 30. Observe the program's behavior. What do you notice?
4. Write a SimCom program that computes the square of the value in byte 31 and stores the result in byte 30. What happens when you try to compute the square of 254?
5. What features could be added to SimCom to make it more useful?

Chapter 2: Data

Overview

Computers process *data*—factual information, such as numbers, text, images, and sound—in a form that can be processed by electronic devices. That is the whole idea of computers. In this chapter, you will see how Java handles data. This chapter will cover the two most important things a program does with data:

- Declaring
- Assigning

Declaring and assigning are activities that we perform in the context of a compiled language such as Java. This chapter will begin by explaining what a compiled language really is. If you are already familiar with this topic, feel free to skip to the [next section, "Data Types."](#)

In the [previous chapter](#), we looked at the SimCom virtual machine and experienced its benefits and drawbacks. SimCom was not much of a computer, but it was valuable as a learning tool. The drawbacks mostly had to do with scale: SimCom did not have enough memory or commands to do anything very interesting. In this chapter, we leave SimCom behind and discuss Java itself.

Note Jumping into Java can be difficult if you're learning programming from the ground up. Even the simplest possible Java program uses many unfamiliar constructs, including classes, methods, arrays, access, and static code. We can't expect you to learn all these concepts before you look at a Java program. So beginning in this chapter, you will be asked to accept that certain parts of all Java programs have to be in place in order for the program to work at all. Eventually, later chapters will present everything you are being asked to accept.

Understanding Compiled Languages

The SimCom virtual computer is difficult to program. You have two options for specifying an instruction: You can enter a byte value, or you can specify an opcode and an address. You've probably found that specifying an opcode and an address is a much better approach, but it's still not very intuitive.

The language of programming with opcodes is known as *assembly language*. Every line of an assembly language program roughly corresponds to a computer instruction in memory. Real computers have much more memory than SimCom, and assembly programs that make real machines do something useful can be quite long. Such programs are created using a text editor. The resulting file is known as *source code*, and it must be translated into the appropriate binary values before it can be executed by a computer.

Conceivably, this translation could be done by people. In fact, in the very earliest days of programming, that's how it was done. However, computers can do a much better job of it. Any program that translates assembly code into computer base-2 code is called an *assembler*. [Figure 2.1](#) shows the flow from assembly language source code to executable computer code.

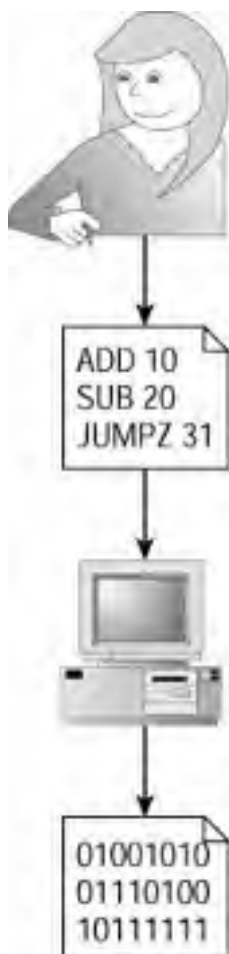


Figure 2.1: Assembly language

After a program has been assembled, it must be loaded into memory and the computer must be told to execute it. This is the job of the operating system.

Assembly programming has many shortcomings, all of which result from being too close to the underlying architecture. When you are forced to think in terms of the interrelationships among hardware components, it is difficult to also consider the domain of the problem you are trying to solve. For example, if you want to write a program to model weather patterns, you will be better off thinking about air currents and water vapor, not about opcodes and registers. To do that, you need a compiled language.

Compiled vs. Assembly Languages

A *compiled language* is like assembly language in the sense that a source program is created using a text editor, and the source must be translated into computer binary. The difference is that, unlike assembly code, a line of source code generally does *not* correspond to a single instruction. In fact, one great benefit of compiled languages is that you don't need to know anything at all about the underlying hardware.

[Figure 2.2](#) shows the flow from compiled language source code to executable computer code.

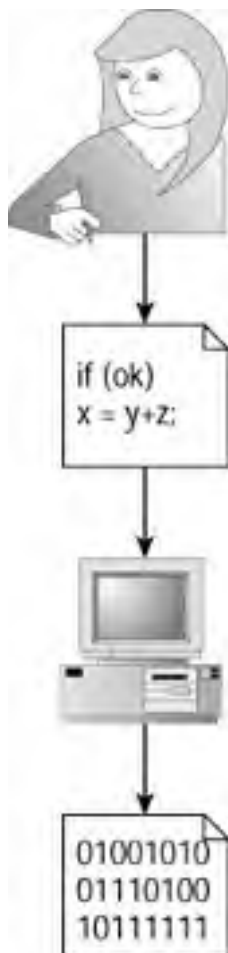


Figure 2.2: Compiled language

Each type of computer (Pentium or SPARC, for example) has its own instruction set and architecture, and hence its own assembly language. However, a compiled language can run on any target machine, provided there's a compiler that can translate it into the target machine's instruction set. For example, there are compilers that translate C++ into Pentium code, and other C++ compilers that produce code for SPARC processors.

Software can be developed much more efficiently with a compiled language than with assembly language. Moreover, in theory a company only needs to develop one version of a software product. When the product is finished, one compiler can be used to produce PC code, another compiler can be used to produce Macintosh code, and so on.

That's the theory. In practice it doesn't work so well. Certainly compiled languages are phenomenally more efficient for development than assembly languages. However, the ideal of developing once and compiling many times is just an ideal. There are differences among target computers that should be negligible, but are in fact significant. Source code that runs flawlessly on one platform may require considerable tweaking to run on a different platform. Multiple versions of source code have to be maintained. The process can get extremely expensive.

The Java Virtual Machine

Java is an *interpreted compiled language*. This means the compiler does not generate code that is specific to any particular processor. Instead, the compiler generates code for an imaginary processor: a *virtual machine*. The compiler does almost all the work. It checks for grammatical correctness, analyzes the structure of the source code, and breaks the source down into elementary units. It does everything except create code that can be run by a computer that exists in the physical world. The Java compiler's output is called *bytecode*, which is the binary format that is understood by the *Java Virtual Machine*, or *JVM*.

The JVM is a program that executes bytecode instructions. Like SimCom in [Chapter 1](#), the JVM's architecture is usually implemented in software rather than being built from circuit components. The JVM itself runs on physical hardware, so there is one version for Windows platforms, one for SPARC platforms, one for Mac platforms, and so on.

When you run a Java application, you are really running the JVM, which in turn loads and executes the bytecode for your application. All JVMs for all platforms execute bytecode in the same way. This means that with Java, you do not have to maintain different versions of source code for different platforms. One of the Java slogans is, "Write once, run anywhere." And it works. With Java, a program has exactly one version of source code. The result of compiling the source—the bytecode—will run on any platform for which a JVM is available.

[Figure 2.3](#) shows the evolution of a Java application from source code through execution.

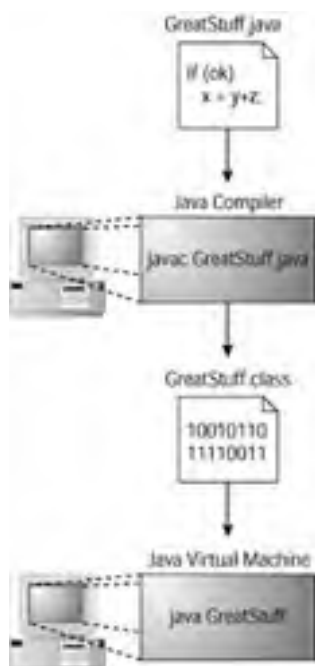


Figure 2.3: Evolution of a Java application

In [Figure 2.3](#), the source code is the file `GreatStuff.java`. All Java source files have to end with `.java` or the compiler won't touch them. The compiler produces one or more files of bytecode output. The bytecode files, also known as *class files*, always end with `.class`. To run a Java program, you type `java classname`, where `classname` is the name of the class file that contains the starting point of the program. Note that here you omit the `.class` suffix. `java` is the name of the JVM program which will read and execute the bytecode class file.

Now that you've seen how the Java compiler and Virtual Machine fit into the big picture, it's time to get acquainted with a fundamental concept of Java programming: data types.

Data Types

Imagine what would happen if SimCom accidentally treated bytes of data as if they were instructions, or instructions as if they were data. In the first case, the virtual machine would execute a random series of opcodes, producing nothing of value. In the second case, instructions would likely be modified (added to or subtracted from one another), again producing nothing of value.

The point is that SimCom uses memory for two different purposes, instructions and data, and each memory type must be treated appropriately. There are no facilities built into SimCom to guarantee appropriate treatment. You just have to be a careful programmer.

This distinction between memory uses is also found in Java and all other high-level languages. Fortunately, Java makes it impossible to execute data or do arithmetic on opcodes.

SimCom has no facilities for dealing with fractions, characters, or very large numbers, and negative numbers are mysterious. Java supports all these different characteristics of numbers. It does this by providing different *data types*. For now, you can think of a data type as a way of using memory to represent data. SimCom uses an eight-bit base-2 representation. Java provides several base-2 representations: two representations for numbers that might contain fractions, one for characters, and one for logical (true/false) values.

Processing a Java data type as if it were a different type would produce worthless results. Java protects you from this kind of problem by requiring you to declare all data types; the compiler enforces the integrity of your declarations. Of course, this will make much more sense later in this chapter, after we discuss declarations. Right now, let's look at Java's data types. Later on, you'll see how they're used.

Integer Data Types

In the terminology of programming, an *integer* is a data type that represents non-fractional numbers. In Java, all integer types are *signed*, meaning that both positive and negative values are supported (as is zero). Java's four integer types are shown in [Table 2.1](#).

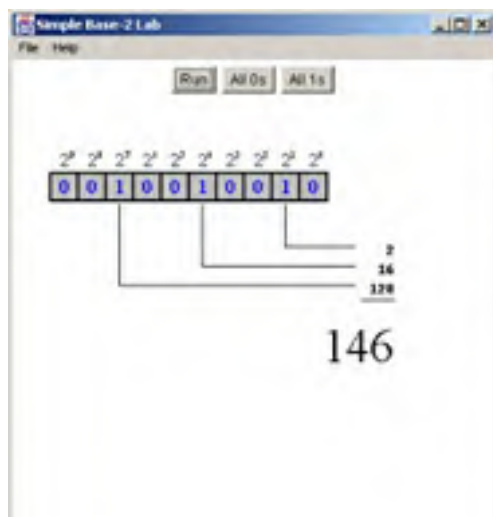
Table 2.1: Java's Integer Data Types

Name	Size	Minimum Value	Maximum Value
byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807

Each data type shown in [Table 2.1](#) has a finite range. Wider ranges are accommodated by data types that require more memory. No type is unlimited – each has a minimum and a maximum value – but it is difficult to imagine exhausting the capacity of the `long` type, which ranges from minus nine quintillion to plus nine quintillion.

Java uses a format known as *two's complement* to represent negative numbers. In nearly all cases, the details of this representation are hidden from programmers, so you can go for a long time without having to know about it. However, there are times when a program will produce baffling results if you don't know about two's complement. Also, there are some arithmetic operators (discussed in [Chapter 3, "Operations"](#)) that only make sense if you know how negative numbers are represented.

Two's complement is an evolution of the classical base-2 notation that we all learned in elementary school. If you need a review, you can run the Simple Base 2 animated illustration on the CD-ROM. First run the Java setup script you created in [Appendix A](#) (assuming you haven't run it already), and type `java twoscomp.SimpleBase2Lab`. You see a ten-bit number. You can click on individual bits to change their values. When you're ready, click the Run button to see which number is represented. [Figure 2.4](#) shows SimpleBase2Lab in action.



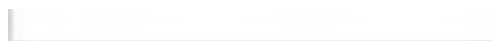


Figure 2.4: SimpleBase2Lab

Straightforward base-2 notation, as shown in the Simple Base 2 animation, is not exactly what computers use to represent numbers. Two's complement is more sophisticated than regular base-2, because of the way negative numbers are represented.

Imagine a car with an odometer that uses base-2 rather than base-10. There are a lot more digits than usual, and they roll over more frequently, but otherwise this odometer is like an ordinary odometer. Every time you drive another mile, the displayed number increases by 1. When the display is showing all 1s, and you drive one more mile, the odometer rolls over and shows all 0s. Thus if you wanted to get imaginative, you could say that in a way a display of all 1s represented -1 mile, because when you add one more mile, you get zero miles.

What about a display that consists of all 1s except for the rightmost digit, which is zero? (This would be 11111110 on an 8-bit odometer.) You could make a case that this reading represents -2 miles, because when you drive two more miles you get zero miles.

Here is another way to make the same case: if you were willing to break the law, you could open the odometer and roll it back manually. If it initially showed one mile and you rolled it back once, it would show zero miles. If you then rolled it back once more, it would show 11111111.

Figure 2.5 shows a base-2 odometer.

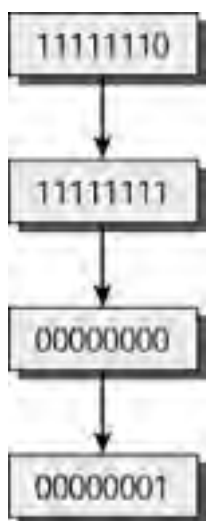


Figure 2.5: A base-2 odometer

Two's complement works like an odometer. A value of all 1s represents -1. Other values are assigned to ensure consistency. For example, with an 8-bit byte, a value of 11111110 represents -2. This makes sense, because adding 1 produces the "all 1s" representation for -1.

The general rules for two's complement are as follows:

- A value of all 0s represents zero.
- If the leftmost bit is 0, the number is positive. The remaining bits represent the value in base-2.
- If the leftmost bit is 1, the number is negative. To compute the magnitude of the value, invert all bits (changing 0s to 1s and 1s to 0s) and then add 1.

Figure 2.6 shows how to compute the value of the 16-bit short 1111111110011001.

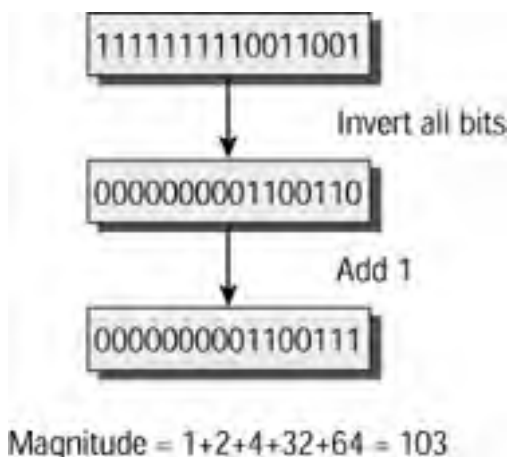


Figure 2.6: An example of two's complement

Figure 2.6 demonstrates that after you invert all the bits and add 1, the magnitude is 103. Thus, the original value of 1111111110011001 must represent -103.

Thinking in two's complement is not intuitive, but fortunately you rarely have to do it. However, it is important to get familiar with this format. There is an animated illustration on the CD-ROM to make this process more enjoyable. To run it, type `java twoscomp.TwoCompLab`. Figure 2.7 shows the program.

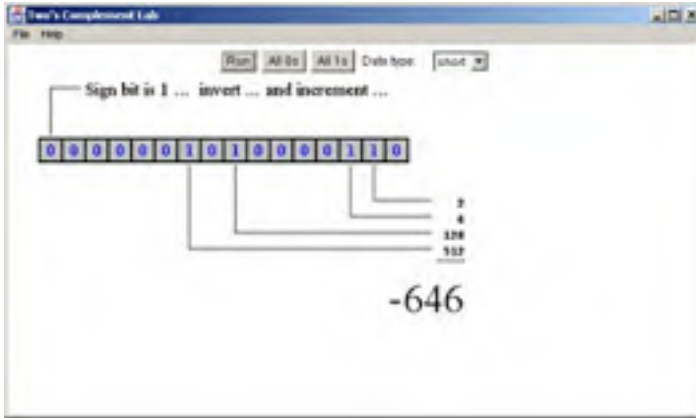


Figure 2.7: Two's complement lab

You can select 8-, 16-, or 32-bit data, corresponding to Java's byte, short, and int data types. (The 64-bit long type does not fit on a screen.) Buttons allow you to set the data to all 0s or all 1s. You can click on an individual bit to change its value. When you are ready, click on the Go button. The program will animate the steps involved in computing the value represented by the bit pattern.

Compute the value represented by the "all 1s" pattern for the byte, short, and int types.

Floating-Point Data Types

Integer data types cannot represent fractions. If you try to use an integer type to store a number with a fractional part, the fractional part will just be discarded. For example, if you divide 29 by 10 (as you'll do in the [next chapter](#)) and store the result in a short, you will find that the short contains 2, not 2.9.

Floating-point data types can represent numbers with fractional parts. Java provides two floating-point data types, called *float* and *double*, as shown in [Table 2.2](#).

Table 2.2: Java's Floating-Point Data Types

Name	Size	Minimum Value	Maximum Value	Smallest-Magnitude Positive Value
float	32 bits	-3.4×10^{38}	3.4×10^{38}	1.4×10^{-45}
double	64 bits	-1.8×10^{308}	1.8×10^{308}	4.9×10^{-324}

The maximum value for a float is approximately 34 followed by a string of 37 zeros: 340 undecillion. With such magnitudes, the common ways of naming numbers become impractical. We use *scientific notation*, as shown in the value columns of [Table 2.2](#). With scientific notation, a number that would ordinarily have a huge string of zeros is represented by a value between 1 and 10 (always strictly less than 10), multiplied by 10 raised to the appropriate power.

The rightmost column of [Table 2.2](#) shows the smallest positive numbers that the data types can represent. These values contain long strings of zeros, not because they are very large, but because they are very small. 1.4×10^{-45} is another way of saying 0.00014.

The original computer output devices—terminals and teletypes—only had one font size, so superscripted exponents could not be displayed. An abbreviation known as *scientific notation* was developed. In scientific notation, the letter E (which is short for *exponent*) is shorthand for "times-ten-to-the." For example, the scientific notation for 3.45×10^{-67} would be 3.45E-67. If you write code that prints out large numbers or very small fractions, you are likely to see scientific notation.

In the discussion of integer data types, you learned that you must understand two's complement notation to really understand certain Java operations. Fortunately, you don't need to know how floating-point numbers are represented internally to understand any Java operations. However, if you are interested in how this is done, you can run the Floating-Point Lab animated illustration by typing `java floating.FloatFrame`. The program lets you vary the bits of a 32-bit `float` number and observe the effect this has on the value. As you might expect, the fractional and exponent parts of the data are in base 2, and the exponent is a power of 2 rather than 10. You might discover certain bit combinations that result in "special values".

Doubles use 64 bits, twice as many as floats. The extra bits are used to give the data type both more range and more precision. The name *double* originates from *double precision*.

Representing Characters

Java uses a 16-bit data type called `char` to represent text characters. The data type can accommodate 2^{16} or 65,536 bit

Java uses a 16-bit data type called `char` to represent text characters. The `char` type can accommodate 2^{16} or 65,536 bit combinations, so 65,536 characters can be represented. This is more than enough to encode all European-based languages, but not enough for Chinese, Japanese, Korean, and certain others. The correspondence between characters and bit combinations is defined by the [Unicode](http://www.unicode.org) standard, which is beautifully described at www.unicode.org.

Representing Logical Values

The integer and floating-point formats represent numerical values. The `char` data type represents text characters. Java has one last data type, *boolean*, which represents logical values. Different JVMs may use different numbers of bytes to store booleans. Often 4 bytes are used, although this is not always the case.

The numerical and `char` types can represent many different values, from 256 possible values for byte all the way up to 18446744073709551616 for long. The `boolean` type can represent only two possible values: `true` and `false`. This data type is useful for controlling conditional execution. For example, a block of code might need to execute only if it's midnight and a certain database query returns more than 100 records but less than 500. Or a block of code might need to execute if the user has entered a special request and a password. Java uses logical values to express conditions like these that might be true and might be false. As you will see in [Chapter 3](#), there are special boolean operations that operate on these values.

Logical values and the operations that act upon them were first studied by George Boole, an 18th-century British mathematician. He is the only person in history whose name has been immortalized as a computer-language concept.

Recap of Java's Data Types

So far this chapter has introduced Java's 8 basic data types. These types are summarized in [Table 2.3](#).

Table 2.3: Java's Primitive Data Types

Data Type	# of Bits	Used For	Internal Format
byte	8	Very small integers	2's complement
short	16	Small integers	2's complement
int	32	Integers	2's complement
long	64	Large integers	2's complement
float	32	Fractions, very large numbers	Floating-point
double	64	Fractions, huge numbers	Floating-point
char	16	Characters	Unicode
boolean	??	Logic	Unavailable

These data types are collectively called *primitives* to distinguish them from object-oriented types. (We will begin our study of objects in [Chapter 7](#).)

We now turn to the question of what you can do with all this data.

Declaring and Assigning

In a sense, computer programming is the art of assigning the right value to the right data at the right time. In Java, as in many other languages, you have to declare your data before you use it. *Declaring* means telling the compiler the types of data you will be using. In this section you will see how to declare and assign data, and will look at your first complete Java program.

When you programmed SimCom, you had to specify the address of each data operand. That meant you had to remember what you were using the different memory bytes for. For example, the Times 5 program used byte #29 as a loop counter and byte #30 for storing the result. Yet, when you look at the program for the first time, it's very difficult to tell what's going on.

In Java, you never have to remember which memory location is being used for which purpose. In fact, there is no way to even *know* which memory location is being used for which purpose. You pick a name for each memory location you want to use, and you refer to memory locations by name rather by address. The compiler assigns the addresses. All you have to do is tell the compiler the names you will be using, and the data type associated with each name.

For example, if you wanted to use a byte as a loop counter, it would be reasonable to choose the name `loopCounter`. Then you would declare as follows:

```
byte loopCounter;
```

A piece of memory that is declared and named in this way is known as a *variable*, so we will use that term from here on.

A declaration has three parts: a data type, a name, and a semicolon.

The data type (for now) is one of the eight primitive types: byte, short, int, long, float, double, char, and boolean. Later we will introduce some other types.

The name has to begin with a letter, an underline (`_`), or a dollar sign (`$`). The rest of the name can consist of letters, underlines, dollar signs, or digits. It is good programming practice to use variable names that begin with lowercase letters. If the name consists of more than one word, the second word and all subsequent words begin with uppercase letters. This is what we have done with `loopCounter`. Later in this book, you will see that there are other entities besides variables for which you will assign names (including classes and interfaces). These entities use different naming conventions. Following the conventions helps make source code easy to read.

The semicolon is a vital part of a declaration. A declaration is a kind of *statement*. A statement is a single instruction. All statements must end with a semicolon. Otherwise, the compilation will fail and the compiler will print out an error message with the line number where it ran into trouble.

Be aware that it is inherently impossible to create a compiler that produces consistently helpful error messages. Imagine someone running along a rough cobblestone road. If his foot slips on a stone, he might stagger for a few steps before falling. Similarly, if the compiler slips on an ungrammatical line, it might stagger over a few more lines before crashing and printing a message.

For the sake of convenience, you can declare multiple variables in a single statement, as long as the variables are all of the same type. So the following:

```
double mass, velocity, energy;
```

is equivalent to the following:

```
double mass;
double velocity;
double energy;
```

After you declare a variable, you can assign values to it. The following two lines declare and assign a variable called `velocity`:

```
double velocity;
velocity = 123.456;
```

Notice that the assignment statement, like the declaration statement, ends with a semicolon. An assignment statement has the form `variable = value semicolon`. (In the [next chapter](#), you will see how the value can be a complicated mathematical formula. For now, the value will be a simple literal number.) Be aware that the equal sign is just a symbol, and its meaning is not exactly the same as its meaning in a mathematical context. In geometry, when we say $\text{Area} = \pi r^2$, the equal sign means "is, always has been, and always will be." In Java, the

equal sign means "store the value to the right of the equal sign in the variable to the left of the equal sign."

When you assign to a char variable, the easiest approach is to enclose the value in single quotes, like this:

```
char ch;
c = 'w';
```

After execution, the variable `ch` contains the Unicode representation for the letter `w`. The single quotes can also contain special codes, called *escape codes*, that encode special characters. The most useful of these are

- `'\n'` – Newline
- `'\t'` – Tab

A Very Simple Java Program

So far we have seen declaration and assignment lines, but only as code fragments. If you type any of the fragments into a file and try to compile the file, you will get nothing more than compiler error messages. This is because a well-formed Java program—even one that does almost nothing—must conform to certain structural rules.

And here we have a problem, which is best illustrated by an example. The following code listing is a complete Java program that contains a declaration and an assignment.

```
public class VerySimple
{
    public static void main(String[] args)
    {
        double age;
        age = 123.456;
    }
}
```

The problem is this: the program contains a number of words and symbols that have not yet been introduced, and that will require considerable explanation when the time comes. So for now, you just have to accept the mysterious parts of this code as things that must be done to make the program work.

Type the program into a file called `VerySimple.java`. Compile it by typing `javac VerySimple.java`. If you get compiler error messages, make sure you've typed in the program exactly as it appears here. The compiler output will be a file called `VerySimple.class`.

The preceding program appears just as it would in a source file. However, when listings of more than a few lines appear in print, it is convenient to number the lines:

```
1. public class VerySimple
2. {
3.     public static void main(String[] args)
4.     {
5.         double age;
6.         age = 12.34;
7.     }
8. }
```

The line numbers are convenient for referring to features of the code, but they should never appear in source code that is to be compiled. Here, the line numbers let us point out that the relevant parts of the listing are lines 5 and 6, and all the rest is mysterious code that will be explained later.

Output

The SimCom virtual machine lets you see all of memory all the time, but Java's memory is hidden. In the `VerySimple` program, there is no way to see the value of `age`. The following program declares and assigns `age`, and then prints its value to the console:

```
1. public class VerySimple2
2. {
3.     public static void main(String[] args)
4.     {
5.         double age;
6.         age = 12.34;
7.         System.out.println(age);
8.     }
9. }
```

The new line is #7. To print out any value, you can use the statement `System.out.println(theValue);`.

Here again, we ask you to accept that the syntax works. The explanation of *why* it works will come as soon as we have covered all the underlying concepts. For now, be aware that in order to print the value of any variable, you need to type that variable's name between the parentheses in a line like #7.

Notice that in line #1, the word following `class` has been changed from `VerySimple` to `VerySimple2`. The name following `class` has to match the name of the source file. Therefore, if you want to type in this program, you should store it in a file called `VerySimple2.java`. The compiler will generate an output file with the same name, followed by the `.class` suffix: `VerySimple2.class`. This compiler-output file is known as a *class file*. To run the application, type `java VerySimple2`. [Table 2.3](#) summarizes this naming consistency.

Table 2.3: Naming Consistency

Name in class line	Source filename	Class filename	Invocation
<code>VerySimple2</code>	<code>VerySimple2.java</code>	<code>VerySimple2.class</code>	<code>java VerySimple2</code>

Printing out the value of a variable is convenient, but it would be even more convenient to print out a reminder of what the value represents. "Age is 12.34" is much more informative than "12.34." The following program prints out the more informative line:

```
1. public class VerySimple3
2. {
3.     public static void main(String[] args)
4.     {
5.         double age;
6.         age = 123.456;
7.         System.out.println("Age is " + age);
8.     }
9. }
```

In line #7, the text inside the double quotes is known as a *literal string*. The plus sign does not indicate addition, since adding text to a number doesn't really mean anything. In this context, the plus sign just means that the literal string is to be printed out, followed by the value of `age`. Within the parentheses of a `println` statement, you can have any number of alternating literal strings and variables. So if you wanted to print out the values of variables `i`, `j`, and `k`, separated by commas, you could use the following line:

```
System.out.println(i + "," + j + "," + k);
```

Now that you can declare, assign, and display variables, you are ready for the next step: mathematical operations. That is the topic of the [next chapter](#).

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. According to [Table 2.1](#), the maximum values for the byte and short data types are 127 and 32767, respectively. Use the Twos-Complement Lab animated illustration to verify this. Which byte and short bit patterns produce the maximum values? In general, which bit pattern produces the maximum value for a two's complement number of N bits?
2. According to [Table 2.1](#), the minimum values for the byte and short data types are -128 and -32768, respectively. Use the Twos-Complement Lab animated illustration to verify this. What byte and short bit patterns produce the minimum values? In general, what bit pattern produces the minimum value for a two's complement number of N bits?
3. Launch the Twos-Complement Lab animated illustration by typing `java TwosCompLab`, set the data type to int, and set all the bits to 1. Then set the three bits on the right to 0. Compute the value. Do the same for the byte and short data types. What do you observe?
4. Launch the Floating-Point Lab animated illustration by typing `java floating.FloatFrame`. Set the rightmost bit to 1 and all other bits to 0. The value represented is 1.4E-45. Try changing various bits' values by clicking on them. Can you create a value that is smaller than 1.4E-45 but still greater than 0?
5. Write a Java application that declares and assigns values to three int variables named `x`, `y`, and `z`. Print out all three values, separated by commas, on a single line.
6. *White space* means spaces, tabs, and line-break characters. Type in the VerySimple application from this chapter (reproduced below) and experiment with inserting white space. Does anything change during compilation or execution if you insert extra spaces between `public` and `class`? What if you insert a line break between `public` and `class`? Can you find any adjacent words or symbols such that inserting white space between them changes compilation or execution?

```
public class VerySimple
{
    public static void main(String[] args)
    {
        double age;
        age = 123.456;
    }
}
```

Chapter 3: Operations

The [previous chapter](#) showed you how to declare various types of primitive variables, and how to assign and print out the values of variables. This chapter will look at the computational operations that you can perform on Java data. For numeric data—that is, for all primitive types other than `boolean`—these computations include the familiar arithmetic operations of addition, subtraction, multiplication, and division, as well as some more exotic operations. Arithmetic operations are not applicable to `boolean` data, which has its own group of operations.

Before covering these topics, we will look at two ways to make programs easier to read: white space and comments.

White Space and Comments

This chapter is going to present some techniques for writing long and intricate programs. Before we begin, though, let's look at how to write programs that are easy to read and understand.

Consider the following lines of code:

```
int x;  
x = 5;
```

As far as the Java compiler is concerned, this code is identical to the following:

```
int x;      x=5;
```

These two versions produce exactly the same compiled bytecode. In the preceding line, the gap after the semicolon can be created by adding a few tabs or a lot of spaces. Either way, it doesn't matter to the compiler.

The compiler ignores any blank space created by using the spacebar or by typing Tab or Enter. Such space is called *white space*. You can use white space to make your source code more readable. For example, the following code declares four variables:

```
double velocity; boolean b;  
short x;  
  
long  
  
hotSummer;
```

If you use this code in a program, someone who is unfamiliar with the program will have a hard time figuring out what your intention is. (And the person sweating over your source code could be yourself, reviewing your own code long after you originally wrote it.)

You can make your code more readable to humans by manipulating white space, like this:

```
double velocity;  
boolean b;  
short x;  
long hotSummer;
```

Or better yet:

```
double    velocity;  
boolean   b;  
short     x;  
long      hotSummer;
```

In the first example, some spaces and a return have been removed, and a return has been added, so that all the left edges line up. In the second example, white space has been added. Now the eye of any reader, including you, will subconsciously arrange the code into two columns. The words on the left are all data types, and the words on the right are all variable names. The columns are easily aligned: You just press Tab before each data type and before each variable name. This formatting scheme is so clear that it is considered correct style by convention. Any other formatting arrangement would be less readable, and would also be considered sloppy style.

The [previous chapter](#) presented the following simple program:

```
1. public class VerySimple  
2. {  
3.     public static void main(String[] args)  
4.     {  
5.         double age;  
6.         age = 12.34;  
7.     }  
8. }
```

Note Remember that the line numbers are not part of the source code. They are just included to make it easier to refer to particular lines.

A Java program consists of one or more class definitions (though we will not discuss class definitions until [Chapter 7, "Introduction to Objects"](#)). For now, be aware that lines 2-8 are the definition of a class named `VerySimple`. The class definition begins with an open curly bracket (line 2) and ends with a closed curly bracket (line 8). Since these brackets are vertically aligned in the same column, our brains notice that they are spatially related, and we subconsciously assume that they must also be functionally related.

A class definition can contain (among other things) a number of method definitions. Methods will be discussed in detail in [Chapter 4, "Methods"](#); for now, you just need to be aware that lines 4-7 are the definition of something called a method, whose name is `main`. The method definition begins with an open curly bracket (line 4) and ends with a closed curly bracket (line 7). Again, the vertical alignment of the brackets gives us visual information about the structure of the program.

Notice how easy it is to look at lines 3 and 4, which tell us, "the method starts here," and find the end of the method. Within the method, all the code (lines 5 and 6) is vertically aligned. If you look at the listing with your eyes out of focus, all you see are several levels of nested blocks of blurry stuff. A Java program is (mostly) a block that contains blocks that contain blocks, etc. It is extremely important to use white space and indentation to indicate the nesting level of all your lines of code.

In addition to white space, the Java compiler also ignores *comments*. There are two kinds of comments: single-line and multi-line.

A single-line comment begins with two slashes (`//`). There can't be anything between the slashes. The compiler ignores everything from the slashes through the end of the line. This lets you put descriptive text after the slashes. Usually, the text explains what just happened in the line. For example:

```
float distance; // Units are microns
double weight; // Units are ounces
```

Note the use of white space to vertically align the comments.

A multi-line comment, also known as a *traditional* comment, can span more than one line but doesn't have to. This kind of comment begins with a slash immediately followed by an asterisk (`/*`). The comment ends with an asterisk immediately followed by a slash (`*/`). For example:

```
/* Declare and initialize variables that
will later be used for computing
time-distortion effects at relativistic
speeds. All distance units are miles,
not kilometers. */
```

```
double speedOfLight;
int numberOfPlanets;
```

```
speedOfLight = 186000;
numberOfPlanets = 9;
```

Note the use of blank lines to separate the multi-line comment from the declarations, and the declarations from the assignments.

Now that you know how to make source code easy to read, we can move on to the main topic of this chapter, which is how to write a program that makes your computer actually compute something.

Arithmetic Operations

Java's arithmetic operations fall into two categories: basic arithmetic (addition, subtraction, multiplication, and division), and some more exotic operations such as modulo and shifting. We will begin by looking at the simple operations.

Basic Arithmetic

The following code computes and prints out the sum, difference, product, and quotient of two numbers:

```
public class C3
{
    public static void main(String[] args)
    {
        int    x, y;           // Inputs
        int    sum;           // x plus y
        int    diff;          // x minus y
        int    product;       // x times y
        int    quotient;      // x divided by y

        /* First assign initial values to
           the x and y inputs. */
        x = 12;
        y = 3;

        // Now do arithmetic.
        sum = x + y;
        System.out.println("sum = " + sum);
        diff = x - y;
        System.out.println("diff = " + diff);
        product = x * y;
        System.out.println("product = " + product);
        quotient = x / y;
        System.out.println("quotient = " + quotient);
    }
}
```

Note the use of the asterisk (*) to indicate multiplication. The other three symbols (+, -, and /) are recognizable from standard arithmetic. These symbols (as well as a few others that we'll see later on in this chapter) are known as *binary operators*. Here the word *binary* indicates that the operators work on two numbers at a time, known as *operands*. Most Java operators are binary, but there are several *unary operators* that each take a single operand. There is even a *ternary operator* that takes three operands. (We will postpone discussion of the ternary operator until [Chapter 5, "Conditionals and Loops."](#))

There is nothing surprising about this program's output:

```
sum = 15
diff = 9
product = 36
quotient = 4
```

As mentioned in the [previous chapter](#), the meaning of the equal sign (=) here is a bit different from its traditional mathematical meaning. In Java, the equal sign is called the *assignment operator*. It tells the computer to compute the value on the right-hand side of the equal sign (usually abbreviated *rhs*), and to store the result in the variable that appears on the left-hand side (usually abbreviated *lhs*). Until now, the *rhs* has been a literal value, but as this program shows, the *rhs* can also be a calculation. The calculation's arguments can be variables or literals, so the following lines would be valid:

```
int halfProduct;
halfProduct = product / 2;
```

Java allows you to declare a variable and assign its initial value, all in a single statement. The preceding code can be rewritten as

```
int halfProduct = product / 2;
```

Later you can assign a different value to `halfProduct`. You can reassign values to variables as often as you like. Just don't declare the variable more than once, because the second declaration will cause a compiler error.

The *lhs* of an assignment can appear in its own *rhs*. Consider the following line:

```
x = x + 5;
```

If this were a line of algebra and not a computer-language statement, it would be ridiculous. When you subtract *x* from both sides, you get $0 = 5$. But in Java, it is perfectly legal because the equal sign means assignment. The line says to add the value of *x* plus 5 and store the result back in *x*.

Precedence and Parentheses

Multiple operations can be combined in a single statement. For example, you might use the following code to compute the area of a circle whose radius is known:

```
double area = 3.14159 * r * r; // Pi-r-squared
```

Use caution when combining different operators in a single statement. It would be reasonable to expect the statement to be evaluated left to right, but Java doesn't do it that way. For example, you might expect that after the following line executes, the value of *x* is 502:

```
int x = 1000 + 4 / 2;
```

Actually, *x* is 1002. Java gives multiplication and division higher *precedence* than addition and subtraction. This means that in a

statement such as the one above, any multiplication or

division is performed before any addition or subtraction, even if the addition and subtraction appear first. So the division happens first ($4/2 = 2$), and then the addition ($1000+2 = 1002$).

Note Java has strict evaluation precedence rules that govern all the operations presented in this chapter. The precedence is summarized in [Table 3.6](#).

If you don't like a statement's order of evaluation, as dictated by the precedence of its operators, you can use parentheses. Operations that appear in parentheses have higher precedence than operations that do not. So the following code really does compute a result of 502:

```
int x = (1000 + 4) / 2;
```

Parentheses can be nested, as the following example shows:

```
int x = 1+(2*(3-4)+(5-6)*(7+8));
```

The result is -16.

The EvaluatorLab animated illustration will help you get used to parentheses and operator precedence. To launch the program, type `java eval.EvaluatorLab`. You will see the display shown in [Figure 3.1](#).



Figure 3.1: EvaluatorLab

Type any arithmetic expression into the text field and press Enter. The arithmetic expression can consist of any combination of literal integers, parentheses, and the binary operations `+`, `-`, `*`, and `/`. Click on the Run button to see an animation of the evaluation of the expression. Click on Step to see an animation of just the next step in the expression's evaluation. The Run Lightspeed and Step Lightspeed buttons perform the evaluation immediately, without animation. [Figure 3.2](#) shows the program after evaluating the configuration of [Figure 3.2](#).

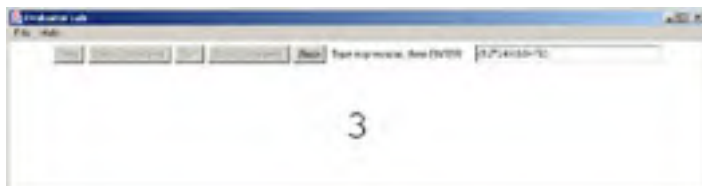


Figure 3.2: EvaluatorLab after evaluation

Type the following expressions into EvaluatorLab and observe the results:

- `1000+4/2`
- `(1000+4)/2`
- `1+(2*(3-4)+(5-6)*(7+8))`

The EvaluatorLab only works with integer data. In Java, integer addition, subtraction, and multiplication behave exactly as you would expect. Division, however, has a problem. Dividing an integer by an integer can produce a non-integer result. Be aware that when Java divides a byte, short, char, int, or long by a byte, short, char, int, or long, the result is *truncated*. This means that any fractional part is discarded. For example, $48 / 10$ would be truncated from 4.8 to 4.

Truncation may seem like a problem, but it really isn't. If you are going to be dividing, and you know that the fractional parts of the results will be important, just use a floating-point data type (float or double) rather than an integer type. A good rule of thumb is to use integer types for quantities that can be counted, such as the number of employees or grizzly bears, and to use floating-point types for things that can be measured, such as weight or speed.

Bitwise Operations

A *bitwise* operation treats its operands as collections of individual unrelated bits, rather than as representations of numbers. You can only perform bitwise operations on integer data. Floats and doubles are not allowed.

There is one unary bitwise operator. Its symbol is the tilde (`~`). It toggles all the bits of its operand, changing all 0s to 1s and all 1s to 0s. [Figure 3.3](#) illustrates the operation `~144`.

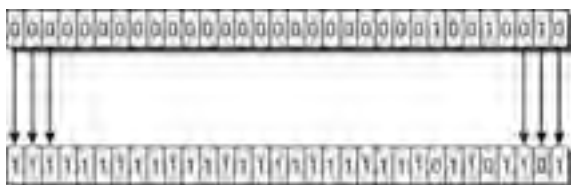


Figure 3.3: The unary bitwise operator ~

With a binary bitwise operation, the *n*th bit of the result is computed from the *n*th bits of the two operands. The three binary bitwise operations are and, or, and exclusive or. The operator symbols are &, |, and ^.

The "and" of two bits is 1 if both bits are 1. Otherwise, the result is 0. Another way to say this is that the result is 1 if one argument bit is 1 *and* the other argument bit is 1.

The "or" of two bits is 1 if either (or both) of the bits is 1. Otherwise, the result is 0. Another way to say this is that the result is 1 if one argument bit is 1 *or* the other argument bit is 1 (or both).

The "exclusive or" of two bits is 1 if either (but *not* both) of the bits is 1. Otherwise, the result is 0.

Table 3.1 shows the results of the three binary bitwise operations on all possible combinations of operand bits a and b.

Table 3.1: Binary Bitwise Operations

	a&b	a b	a^b
a,b = 0,0	0	0	0
a,b = 0,1	0	1	1
a,b = 1,0	0	1	1
a,b = 1,1	1	1	0

As you can see from the table, the only way for & to generate a 1 is if both operands are 1. The only way for | to generate a 0 is if both operands are 0. ^ generates a 1 if its two operands are different.

In practice, the binary bitwise operators work on integer values, not on integer bits. For example, if you take the "and" of two ints, bit 0 of the result will be the "and" of the bit 0s of the two operands. Bit 1 of the result will be the "and" of the bit 1s of the two operands, and so on through bit 31. This is illustrated in Figure 3.4.

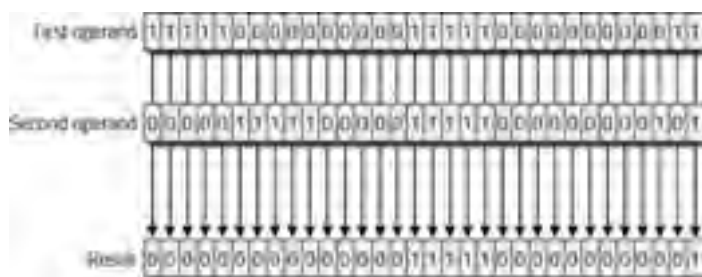


Figure 3.4: Bitwise "and"

As you can see, every bit in the result is computed solely from the corresponding bits in the two operands.

Modulo

Java supports some binary arithmetic operations that we don't often encounter outside the realm of computer programming: the modulo operation and three shift operations.

The symbol for modulo is the percent sign (%). The operation divides the first operand by the second operand and returns the remainder. So for example, 506 % 100 is 6, 507 % 100 is 7, and so on.

Shifting

Shifting operations move the bits of an integer operand to the left or right by some number of positions. There is one left-shift operation; its symbol is <<. There are two right-shift operations; their symbols are >> and >>>.

The left-shift operation is straightforward. The first operand is the value to be shifted. The second operand is the number of bit positions to shift by. Figure 3.5 illustrates 18 << 5.

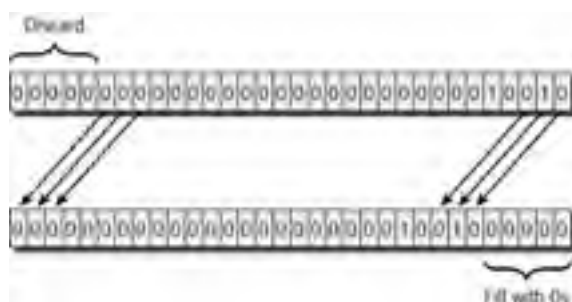


Figure 3.5: Left-shift: <<

As the figure shows, the high-order bits of the shifting value are discarded. The low-order bits are all set to 0.

The result of the shift in base-2 is 1001000000, which is 576. Is there any numerical relationship between 576 and the original value of 18? Yes, 576 is 18 times 32. Is there anything special about 32? Yes, 32 is 2 raised to the power of 5. In general, left-shifting x by y is the same as multiplying x by 2^y . This is elegant, and it makes good sense. Left-shifting a value by, for example, 3 bit positions is like writing three 0s to the right of the number. In base-10, if you write three 0s to the right of a number, you have multiplied that number by 10^3 (that is, by 1000). It is not surprising that something similar happens in base-2.

There are two right-shift operations. One of them is bitwise, and the other is numeric.

The bitwise right-shift (\gg) is just the opposite of the left-shift: bits are moved to the right, any bits that fall off the right end are lost, and the left end is filled with 0s. This is illustrated in [Figure 3.6](#).

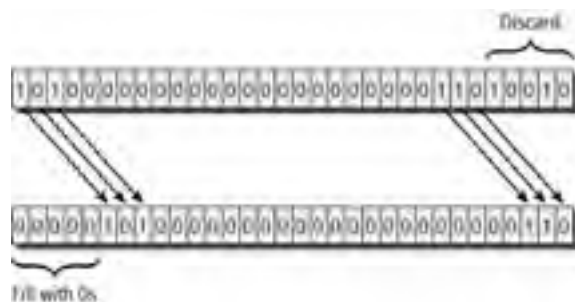


Figure 3.6: Bitwise right-shift: \gg

The original value in the figure has its sign bit set to 1, representing a negative number. The result has a sign bit of 0, since the \gg operation always shifts 0s into the left portion of the result. You can see that \gg always converts negative numbers to positive numbers that have no clear relationship to the original values. This is why the \gg shift is called *bitwise*. All it does is move bits.

The other shift operation is \gg . It is different from \gg in only one respect: The left bits of the result are set to the sign bit of the original value, instead of being always set to 0. For positive numbers, the original sign bit is 0, so \gg is the same as \gg . But for negative numbers, the result is very different, as [Figure 3.7](#) shows.

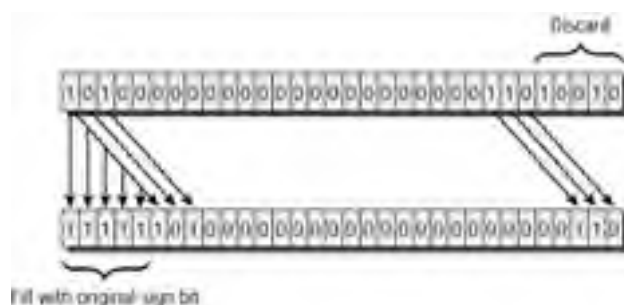


Figure 3.7: Numeric right-shift: \gg

The sign of the result is always the sign of the original value. Does the result have any numerical relationship to the original? Yes, although it is hard to see the relationship when you look at [Figure 3.7](#). It turns out that $x \gg y$ is the same as $x / 2^y$.

The different right-shift operations can be confusing until you have some experience with them. The ShiftLab animated illustration will help you get that experience. Launch the program by typing `java shift.ShiftLab`. The display shows a 32-bit int value to be shifted, as illustrated in [Figure 3.8](#).

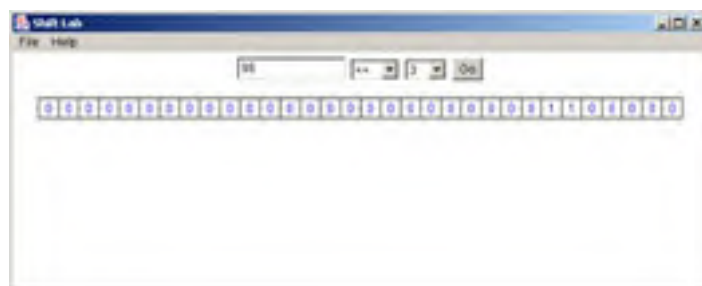


Figure 3.8: ShiftLab

You can change the value by typing a base-10 number (positive or negative) into the text field, or by clicking on individual bits in the display. Select the desired shift operation (\ll , \gg , or \gg) and the desired shift size, and then click on the Go button. The program will animate the shift that you've specified. [Figure 3.9](#) shows the result of "96 \ll 3".



Figure 3.9: ShiftLab after shifting

Try viewing the following shifts:

```
10 << 10
16384 >> 14
-1 >>> 1
-1 >> 1
-1 >> 20
-2147483648 >>> 31
```

Unary Arithmetic

The unary arithmetic operators have the symbols + and -. These are the same as the symbols for binary addition and subtraction, so the compiler has to figure out from context which kind of operation you want. A + or - between two operands is a binary operator; a + or - with no operand to the left is unary.

The unary - operation just changes the sign of its operand. So for example, the following code prints out $y = -5$:

```
int x = 5;
int y = -x;
System.out.println("y = " + y);
```

The unary + operator maintains the sign of its operand. In other words, it doesn't really do anything.

++ and --

Two of the most common operations in programming are adding or subtracting 1 with a variable, and storing the result back in the variable. If the variable is called x , these operations can be programmed as follows:

```
x = x + 1;
x = x - 1;
```

However, Java provides some convenient abbreviations. The first line can be abbreviated in either of the following ways:

```
x++;
++x;_
```

The second line can be abbreviated in either of the following ways:

```
x--;
--x;
```

The following program shows these operators in action:

```
public class PlusPlusMinusMinus
{
    public static void main(String[] args)
    {
        int x = 10;
        int y = x;
        x++;
        y--;
        System.out.println("x=" + x + ", y=" + y);
    }
}
```

The output is

```
x=11, y=9
```

You can see that x has been incremented and y has been decremented.

When the operator appears after the operand, the rhs is first calculated as if the operator were not present. Then the rhs value is assigned to the lhs. Lastly, the operand of ++ or -- is incremented or decremented. For example:

```
public class PostDec
{
    public static void main(String[] args)
    {
        int x = 10;
        int y = 1000 + x--;
        System.out.println("x=" + x + ", y=" + y);
    }
}
```

The output is

$x=9$, $y=1010$

You can see that x , which was originally 10, must have been added to y before being decremented.

When $++$ appears before its argument, it is called the *pre-increment* operator. When it appears after its argument, it is called the *post-increment* operator. Similarly, $--$ before its argument is called the *pre-decrement* operator, and $--$ after its argument is called the *post-decrement* operator.

Team LIB

4 PREVIOUS

NEXT 5

Boolean Operations

So far, all the operations we have looked at have dealt with numbers. Now we turn our attention to operations that work on boolean data. Some of these operations share symbols with similar numeric operations (`|`, for example). However, the boolean versions are essentially different from their numeric counterparts.

Most of Java's boolean operations are binary, and both operands must be of boolean type.

And, Or, Exclusive Or, Inversion

We have already seen these as bitwise arithmetic operations. The symbols for and, or, and exclusive or are, as before, `&`, `|`, and `^`, respectively. The symbol for inversion is `!` rather than `~`.

The following program prints out the results of applying these operators to `true` values:

```
public class BooleanOps
{
    public static void main(String[] args)
    {
        boolean a = true;
        boolean b = true;
        boolean x = a & b;
        System.out.println("true&true = " + x);
        x = a | b;
        System.out.println("true|true = " + x);
        x = a ^ b;
        System.out.println("true^true = " + x);
        x = !a;
        System.out.println("!true = " + x);
    }
}
```

The output is

```
true&true = true
true|true = true
true^true = false
!true = false
```

Boolean operations, like arithmetic operations, have precedence. The unary `!` operator is evaluated before the binary `&`, `|`, and `^`. For example, the value of `!false|true` is `true`, because `!false` is evaluated first. You can override the effects of precedence by using parentheses. In the current example, if you want the `|` operator to execute before `!`, use the expression `!(false|true)`.

The BoolLab animated illustration demonstrates the evaluation of boolean expressions. Launch the program by typing `java bool.BoolLab`. [Figure 3.10](#) shows the program just after it starts up.

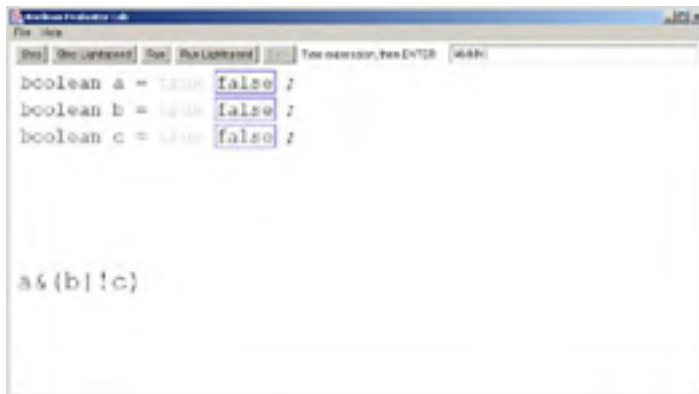


Figure 3.10: BoolLab: initial screen

You can type into the text field any valid expression composed of the variables `a`, `b`, and `c`, the literals `true` and `false`, the operators `&`, `|`, `^`, and `!`, and parentheses. After you enter the expression you want, press Enter. The expression will appear in large font in the main area of the window. As with EvaluatorLab, you can click on the Run button to see an animation of the expression being evaluated. Click on Step to see an animation of just the next step in the expression's evaluation. The Run Lightspeed and Step Lightspeed buttons perform the evaluation immediately, without animation. [Figure 3.11](#) shows the result of running the configuration of [Figure 3.10](#).



Figure 3.11: BoolLab after execution

Try the following expressions in BoolLab:

```
false | false | false | false | false | true
true & true & false & true & true
false & (((!(true ^true) & (false|true))|false)^false)
true | (((!(false ^ false) & (true | false))| true)^ true)
```

The first expression shows that when you take the or of a number of values, a single true is enough to make the entire result true. The second expression shows that when you take the and of a number of values, a single false is enough to make the entire result false.

Now that you have seen Java's simple boolean operators, let's move on to the short-circuit operators, which shorten the time it takes to execute an operation. Before you read the [next section](#), can you guess the point of the lengthy third and fourth expressions in the preceding code?

Short-Circuit Operators

This really happened to me, and perhaps it has happened to you. When I was a little boy, I was allowed to go out and play if I had made my bed and finished my homework. I didn't mind doing my homework, but I hated making my bed and often I wouldn't do it. When my mother asked if I had made my bed, I would start to say, "No, but I ..." I was going to say that I had done my homework, but my mother would interrupt me. She was a busy person and she had heard all she needed to hear. Our agreement was that I would do two chores. As soon as she knew that I had not done *one* of those chores, there was nothing I could say about the *other* chore to convince her that I had lived up to my part of the agreement.

False & anything is false. When you compute $x \& y$, and x is false, you don't have to spend any time at all on y . You already know the answer.

Consider the following expression, which you were invited to type into BoolLab in the [previous section](#):

```
false & (((!(true ^true) & (false|true))|false)^false)
```

At first glance, this expression looks so complicated that you would not want to figure out its value in your head. But at second glance, once you realize that the expression's form is "false & anything," you don't have to look any further. The value is false, no matter what comes after the &.

Java provides an alternative to the & operator. It is called the *short-circuit &* operator, and its symbol is &&. The short-circuit version stops computing and immediately returns false if its first operand is false. Let's slightly modify the previous example:

```
false && (((!(true ^true) & (false|true))|false)^false)
```

Now the first operator is the short-circuit version. This expression evaluates to the same value as the previous version, but the evaluation takes less time because everything between the outermost parentheses is ignored.

There is also a short-circuit version of the | operator. Its symbol is ||, and it immediately returns true if its first operand is true.

The BoolLab animated illustration supports short-circuit operations. Launch the program again (type `java bool.BoolLab`), and see how it evaluates the following expressions:

```
false && (true|false)
true && (true|false)
false || && (true|false)
true || && (true|false)
```

Java's short-circuit operators allow you to profit from the principle that false-and-anything is false and true-or-anything is true. The amount of profit may seem trivial. In this example, the processing time that's saved by using && could not possibly be more than a microsecond or so. But a short-circuit expression might be executed not once but many times—even many millions or billions of times—so any time savings will be significant.

You will learn how to execute a single expression multiple times when you look at loops in [Chapter 5](#). Moreover, the second operand of the short-circuit operator might be a call to a method that takes minutes or hours to execute. In this case, you definitely do *not* want to process the second operand unless you really have to. The [next chapter](#) will look at methods and method calling.

Now let's look at Java's comparison operators. These are binary operators whose operands can be numeric or boolean. The result type is always boolean.

Comparison Operations

Java's comparison operators always return a boolean value. Most of these operators work on numeric operands, but there are two that can take numeric or boolean operands. [Table 3.2](#) summarizes the comparison operators.

Table 3.2: Comparison Operators

Operator	Meaning	Numeric Operands	Boolean Operands
==	Equals	3	3
!=	Does not equal	3	3
>	Is greater than	3	no
>=	Is greater than or equal to	3	no
<	Is less than	3	no
<=	Is less than or equal to	3	no

Note that the symbol for the equals comparison operator is a double equal sign (==), to distinguish it from the assignment symbol (=).

Comparison operators can be combined with other boolean operators. For example, assuming *w*, *x*, *y*, and *z* are variables of some numeric type, you might use the following expression:

```
w == x | y < z
```

Comparison operators have higher precedence than boolean operators, so the == and < comparisons happen before the | is evaluated. The example can be rewritten as follows:

```
(w == x) | (y < z)
```

The parentheses make the expression clearer without changing the order of computation. Expressions such as this one are most often seen in flow-control statements, which allow you to execute blocks of code repeatedly, or only if certain desired conditions are met. We will look at flow-control statements in [Chapter 5](#).

Compound Assignment

A very common practice is to perform an operation on the value of a variable and then store the result back in the variable. For example, you might want to do the following:

```
x = x - y;
```

For situations like this, Java provides an abbreviation called *compound assignment*. A compound assignment lets you modify a variable (in certain restricted ways) and store the value back in the variable, all in a single statement. Compound assignments have the form *variable op= expression*, where *op* is a binary operation symbol that is immediately followed by `=`.

Table 3.3 summarizes the compound assignment operators. (The table assumes that *b* is boolean and *x* is of some numeric type.)

Table 3.3: Compound Assignment

Operator	Example	Equivalent
<code>+=</code>	<code>x += 5;</code>	<code>x = x+5;</code>
<code>-=</code>	<code>x -= 5;</code>	<code>x = x-5;</code>
<code>*=</code>	<code>x *= 5;</code>	<code>x = x*5;</code>
<code>/=</code>	<code>x /= 5;</code>	<code>x = x/5;</code>
<code>%=</code>	<code>x %= 5;</code>	<code>x = x%5;</code>
<code><<=</code>	<code>x <<= 5;</code>	<code>x = x<<5;</code>
<code>>>=</code>	<code>x >>= 5;</code>	<code>x = x>>5;</code>
<code>>>>=</code>	<code>x >>>=5;</code>	<code>x = x>>>5;</code>
<code>&=</code>	<code>b &= false;</code>	<code>b = b&false;</code>
<code> =</code>	<code>b = false;</code>	<code>b = b false</code>
<code>^=</code>	<code>b ^= false;</code>	<code>b = b^false;</code>

Compound assignments provide no new functionality. They just provide a convenient way to abbreviate.

Numeric Result Type

You have now learned about all of Java's unary and binary operators. Before closing this chapter, you need to learn about more topic: the result type of numeric operations.

Clearly, it would be inconvenient to prohibit operating on mixed types. You may find yourself doing arithmetic on two numbers, one of which might be an int and the other of which might be a float. The next issue to consider is the data type of the result.

Java's rules for determining the type of the result are based on the concept of *width*. As you saw in [Chapter 2](#), every numeric type has a range. This is shown in [Table 3.4](#).

Table 3.4: Ranges of Numeric Types

Name	Size	Minimum Value	Maximum Value
byte	8 bits	-128	127
short	16 bits	-32768	32767
char	16 bits	0	65535
int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807
float	32 bits	-3.4×10^{38}	3.4×10^{38}
double	64 bits	-1.8×10^{308}	-1.8×10^{308}

If the range of any type completely contains the range of another type, the first range is considered to be *wider* than the second range. If a range is completely contained within another range, the first range is *narrower* than the second range. [Table 3.4](#) shows that the byte type is narrower than the short type. Note that some types are neither wider nor narrower than some other types. Short, for example, is neither wider nor narrower than char.

[Figure 3.12](#) illustrates data type width. [Figure 3.12](#) is definitely *not* drawn to scale. If the line representing double were scaled to the line representing byte, the double line would be 5×10^{275} light years long. I really wanted to print the line to scale, because I believe accuracy is important, but my editor pointed out that the line would be 3×10^{273} times the diameter of the universe. The publisher was unwilling to pay for that much ink, and economics won out.

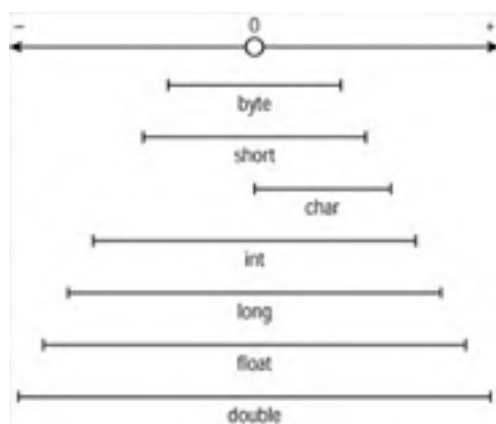


Figure 3.12: Data type width, not to scale

Another way to imagine width is shown in [Figure 3.13](#). A type is wider than another type if you can get from the first type to the second type by following the arrows. So double is wider than byte, and long is wider than char.

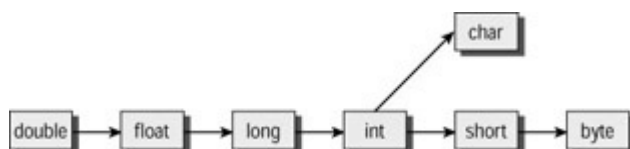


Figure 3.13: Data type width relationships

[Figure 3.13](#) shows that float is wider than long, even though longs are 64 bits and floats are only 32 bits. That might seem backwards, but you'll see why it's true if you think about the definition of "*wider*." If you don't feel like thinking about that right now, you can wait until you get to Exercise 6.

Java's rule for the result data type is this: It's either int or the type of the widest operand, whichever is wider. This means that the result of any arithmetic operation will never be a byte, short, or char.

This rule applies to unary as well binary operations. For example, if s is a short, -s is an int.

Table 3.5 summarizes the result type combinations for binary operations.

Table 3.5: Binary Arithmetic Result Types

	byte	short	char	int	long	float	double
byte	int	int	int	int	long	float	double
short	int	int	int	int	long	float	double
char	int	int	int	int	long	float	double
int	int	int	int	int	long	float	double
long	long	long	long	long	long	float	double
float	float	float	float	float	float	float	double
double	double	double	double	double	double	double	double

It is important to know about arithmetic result types because of another rule: You can only assign a numeric value to a variable whose type is the same as, or wider than, the type of the numeric value. If you try to do anything else, the compiler will generate an error. This makes sense, because you might be trying to store a value that the variable cannot represent. For example, you can't store a long value in a byte variable, because the long value might be greater than 127 or less than -128. So the following code fragment will generate a compiler error:

```
long distance = 999999;  
long time = 5000;  
byte rate = distance / time;
```

This rule sometimes gets in your way when you just want to initialize a variable with a literal value. Java dictates that all floating-point literals are doubles, and all integral literals are ints. So 3.14 and 2.5e33 are both doubles, and 1234 is an int.

If you try to assign a value like 3.14 to a float (such as `float f = 3.14;`), the compiler will complain that you are trying to assign a double to a float. To fix the problem, append the letter `f` or `F` to the end of the literal number. This will tell the compiler that the literal is really a float:

```
float f = 3.14f; // Or 3.14F
```

The situation is a bit stranger if you try to assign a big literal value to a long variable. The following line generates a compiler error:

```
long timeAgo = 999999999999; // 12 digits
```

The 12-digit string of 9s is too big to be represented by an int. Even though you innocently want to assign a big number to a long variable, behind the scenes the compiler is going to try to create an int to store the value 999999999999. This is because the compiler uses ints to store literal integral numbers. To get around the problem, append the letter `l` or `L` to the literal value to indicate that it's really a long:

```
long timeAgo = 999999999999L;
```

You could also use 999999999999l, but a lowercase `l` looks too much like a `1`. The uppercase version is definitely preferable.

What happens when you want to assign a literal value to a byte, short, or char variable? In this case, the compiler gives you a break. As long as the literal value falls within the variable's range, statements such as these are legal:

```
byte x = 12;  
short y = -22;  
char z = 0;
```

Precedence Summary

This chapter presented 16 Java operations. Their evaluation precedence is shown in [Table 3.6](#). Higher-precedence operators, the ones that are evaluated first, appear at the top.

Table 3.6: Operator Precedence

Category	Operators
Unary	+ - ! ~ ++ --
Higher-precedence arithmetic	* / %
Lower-precedence arithmetic	+ -
Shift	>> << >>>
Bitwise	& ^
Short circuit	&&

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. What happens when a comment appears inside a literal string? (Recall from [Chapter 2](#) that a literal string is a run of text enclosed between double quotes.) What would the following line of code do?

```
System.out.println("A /* Did this print? */ Z");
```

Write a program that includes this line. Does the program print the entire literal string, or does it just print "A Z"?

2. What is the value of ~ 100 ? What is the value of $\sim \sim 100$? First try to figure it out, and then write a program to print out the values. (Hint: You can figure it out without using pen and paper if you remember something that was discussed in [Chapter 2](#).)
3. Write a program that prints out the following values:

```
32 << 3
```

```
32 >> 3
```

```
32 >>> 3
```

```
-32 << 3
```

```
-32 >> 3
```

```
-32 >>> 3
```

4. What are the values of the following expressions? First do the computations mentally. Then write a program to verify your answer.

```
false & ((true^(true&(false|!(true|false))))^true)  
true | (true^false^false^true&(false|!(true&true)))
```

5. The following expression looks innocent:

```
boolean b = (x == 0) | (10/x > 3);
```

You can assume x is an int. Write a program that prints out the value of this expression for the following values of x : 5, 2, 0. What goes wrong? (You will see a failure message that you might not be familiar with, because we have not introduced it yet. Don't worry – just try to understand the general concept.) How can you make the code more robust by adding a single character to the expression?

6. The 32-bit float type is wider than the 64-bit long type. How can a 32-bit type be wider than a 64-bit type?
7. Write a program that contains the following two lines:

```
byte b = 6;  
byte b1 = -b;
```

What happens when you try to compile the program?

Chapter 4: **Methods**

Overview

So far, all of the Java applications we have seen have been linear: The processing has proceeded line by line, from the start through the end of the block that begins `public static void main(String[] args)`.

Such applications don't really take full advantage of your computer's capability. In fact, with linear code your computer is not much more than an expensive calculator. The topics in the following two chapters will begin to branch out—and so will the paths of execution through the programs we will study. Instead of proceeding line by line, we will see how to make the execution path detour, fork, and loop.

This chapter will look at methods, which are detours in the path of execution. [Chapter 5, "Conditionals and Loops,"](#) will introduce statements that redirect the flow of the program.

Method Structure

The easiest way to introduce methods is with an example. Suppose you want to print out the fifth powers of the numbers 5 through 9. The following code, which doesn't use methods, does the job in a clumsy, inelegant way:

```
1. public class NoMethods
2. {
3.     public static void main(String[] args)
4.     {
5.         int n = 5;
6.         int n5th = n*n*n*n*n;
7.         System.out.println(n + " >=> " + n5th);
8.         n = 6;
9.         n5th = n*n*n*n*n;
10.        System.out.println(n + " >=> " + n5th);
11.        n = 7;
12.        n5th = n*n*n*n*n;
13.        System.out.println(n + " >=> " + n5th);
14.        n = 8;
15.        n5th = n*n*n*n*n;
16.        System.out.println(n + " >=> " + n5th);
17.        n = 9;
18.        n5th = n*n*n*n*n;
19.        System.out.println(n + " >=> " + n5th);
20.    }
21. }
```

The application's output is

```
5 >=> 3125
6 >=> 7776
7 >=> 16807
8 >=> 32768
9 >=> 59049
```

When you look at the source code, you might get the feeling that life ought to be better than this. The application has a lot of repetition... and aren't computers supposed to be good at eliminating repetitive tasks? Five of the lines (6, 9, 12, 15, and 18) do almost the same computation, but not quite. Each of them multiplies something by itself 5 times.

It would be great to have a piece of subordinate code that could compute the 5th power of *anything*. Of course, there would have to be a way to tell the subordinate code what number to work with. The following application does just that, using methods:

```
1. public class UsesMethods
2. {
3.     public static void main(String[] args)
4.     {
5.         int n = 5;
6.         int n5th = toThe5th(n);
7.         System.out.println(n + " >=> " + n5th);
8.         n = 6;
9.         n5th = toThe5th(n);
10.        System.out.println(n + " >=> " + n5th);
11.        n = 7;
12.        n5th = toThe5th(n);
13.        System.out.println(n + " >=> " + n5th);
14.        n = 8;
15.        n5th = toThe5th(n);
16.        System.out.println(n + " >=> " + n5th);
17.        n = 9;
18.        n5th = toThe5th(n);
19.        System.out.println(n + " >=> " + n5th);
20.    }
21.
22.    static int toThe5th(int x)
23.    {
24.        int result = x * x * x * x * x;
25.        return result;
26.    }
27. }
```

The application's output is the same as the output from the previous version.

The code from lines 22-26 constitutes a method. Line 22 is called the method's *declaration*. It tells the compiler that what is about to follow will be the definition of the method whose name (along with some other information) appears in the declaration line. The definition, or *body*, of a method immediately follows the declaration, and it must appear within curly brackets.

The general format of a method declaration is

Optional_modifiers Return_type Name(Optional_arguments)

The only mandatory parts of a declaration are the return type, the name, and the parentheses. In this example, we have one modifier (static), the return type is int, the method's name is toThe5th, and there is one argument (int x) that appears inside the parentheses. Let's look at each of these elements.

You have already been patiently tolerating the unexplained presence of the *static* modifier in every application we have looked at in this book. It has appeared in the declaration of `main`, which is a method that appears in every Java application.

Understanding `static` will become much easier after we introduce object-oriented programming in [Chapter 7](#). For now, let's just say that `static` means "don't be object-oriented." Other modifiers that might appear in a method declaration include access modifiers, which will be presented in [Chapter 9](#).

We'll look at the return type in a moment. Let's move now to the method name. The rules for the name are the same as those for variable names: The first character must be a letter, an underscore, or a dollar sign. The subsequent characters may be any of these, or they may be digits.

As with variable names, you have a broad choice. You should pick the name that does the best possible job of describing what the method does. When the name appears outside the method, as in lines 6, 9, 12, 15, and 18 of this example, the path of program execution detours through the method. This is known as *calling* or *invoking* the method, and a line that calls a method is the method's *caller*.

Note that in the lines where the method is called, the method call (the name followed by something parenthetical) is used in a context where you would expect to see a value. Until now, the right-hand side of an assignment has been either a literal, a variable, or an arithmetic or boolean expression composed of literals, variables, and operators. Now we add something new to the mix. Anywhere the compiler expects a value, you can use a method call. This is because a method call produces a value, called the method's *return value*. When the computer executes a line of code that includes a method call, the computer takes a detour through the method body in order to compute the return value. When the detour is finished, execution continues where it left off.

The arguments are the method's inputs. In this example, the argument list is `int x`. This means that the method has one input, whose type is `int`. Within the body of the method, that input will be called `x`. When the method runs, the actual value of `x` will be whatever the caller wants. The caller specifies an input value by putting the value in parentheses in the call line. Lines 6, 9, 12, 15, and 18 all pass `n` as the method argument, but the value of `n` is different for each of those lines. In line 6, `n` is 5, so the call from line 6 will execute with `x` set to 5. In line 9, `n` is 6, so the call from line 9 will execute with `x` set to 6. And so on. This demonstrates the flexibility of methods.

The return type in the method declaration tells the type of the return value. The returning of a value happens in line 25, where we see the `return` keyword. This causes the path of execution to return to the caller line. The value following `return` is the return value. In this example, the return value is `result`, which is the 5th power of the argument.

Argument Lists

The `toThe5th` method in the previous example took a single argument, so within the parentheses in the method declaration, we saw a single type (`int`) followed by a single name (`x`). You can create methods with an arbitrarily number of arguments of arbitrary types. To do this, just write a declaration with the following format:

```
Mods Ret_type Name(type0 arg0, type1 arg1, type2 arg2 ...)
```

Note Note that the numbering of the arguments starts at 0, rather than 1. Whenever you see someone do this, you can be certain that they work in the computer field, where counting from 0 is conventional. We already saw this in [Chapter 1](#), where SimCom's memory addresses started at 0. We will see it again in [Chapter 6, "Arrays."](#) It is important to get into the habit of counting from 0 as soon as possible, even though there is a slight inconvenience. When you start counting from 1, the last number you count out is the actual number of things you've counted. When you start from 0, the actual number is 1 more than the last number you've counted out. So in the preceding declaration format, we have 3 arguments, and the largest index is 2.

The following method takes 2 arguments:

```
static double hypotSquared(double leg0, double leg1)
{
    return leg0*leg0 + leg1*leg1;
}
```

Recall that in any arithmetic expression, multiplication takes precedence over addition, so the method really does return the square of the hypotenuse.

The MethodLab animated illustration demonstrates the passing of arguments and the returning of return values. To run the program, type `java methods.MethodLab`. The display presents code that calls a method, as shown in [Figure 4.1](#).



Figure 4.1: MethodLab

Click on the Run button to see the animation. The black lines indicate the passing of arguments, which are called *a* and *b* in the caller but *x* and *y* in the method. The blue line represents returning the return value. When the animation is finished, click Reset to start again. [Figure 4.2](#) shows MethodLab after the animation finishes.

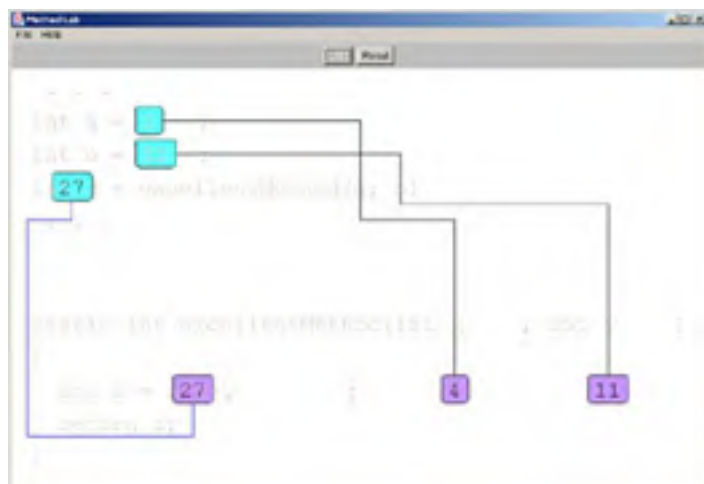


Figure 4.2: MethodLab after animating

You can customize MethodLab by typing any integer value you like into the text fields for *a* and *b*. You can also enter any numeric formula for the value of *z*, which becomes the return value. Try the formula in the preceding example. Since this method's arguments are called *x* and *y*, the formula should be

```
x*x + y*y
```

With this formula, try running MethodLab with *a* = 3 and *b* = 4. Try again with *a* = 12 and *b* = 5. These combinations are the only small integers that represent triangles with integer hypotenuses.

A method's argument list can be arbitrarily long. At times you might even want a method with no arguments at all. In that case, the method's declaration has an empty pair of parentheses after the name, and the caller passes nothing at all inside its own parentheses:

```
float f = sayHello();
...
static float sayHello()
{
    System.out.println("Hello");
    return 3.14159f;
}
```

The important thing is that when you call a method, the call should have the same number of arguments as the method declaration, and the types of the arguments passed by the caller should be compatible with the types in the method declaration. This is subtly different from saying that the caller's argument types should exactly match the types in the method declaration. Recall from [Chapter 3](#) that a value can be assigned to a variable of a different type, provided the new type is wider than the old type. [Figure 4.3](#) (first shown in [Chapter 3](#)) shows the width relationships among the numeric primitive types.

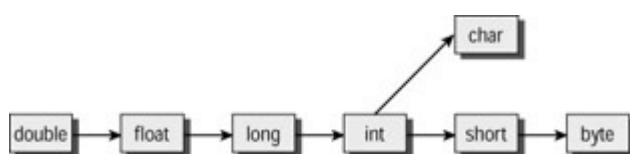


Figure 4.3: Numeric type widths

The rule for passing method arguments is similar to the rule for assignment: *You can pass an argument whose type is different from the type declared by the method, provided the type declared by the method is wider than the type that you pass.* Recall that a type is wider than another type if you can get from the first type to the second type by following the arrows in [Figure 4.3](#).

For example, suppose a method has the following declaration:

```
static char abcde(long wayToGo)
```

This method can be called with a long argument, or with any argument whose type is narrower than long: byte, short, char, or int.

More on Return Types

At times, you might want to create a method that doesn't return anything. The method might print out a message, display a dialog box, or store a value in a file. In cases such as these, it is difficult to think of any value that the method could meaningfully return, and concocting a return value just for the sake of having one would not contribute to the quality of the program. (A basic principle of writing fiction is that every word should contribute to developing the plot or developing the characters. We can invent a similar principle for writing software: Every word of source code should contribute to the operation or the readability of the program.)

Suppose you want a method that prints a number along with a message. Since no return value is needed or relevant, replace the return type in the declaration with the word `void`:

```
static void printPretty(int x)
{
    System.out.println("x = " + x);
}
```

Note the absence of a `return` statement. A void method runs until it executes its last line. Then it returns automatically. Optionally, you can add a `return` statement with no value at the end of the method:

```
static void printPretty(int x)
{
    System.out.println("x = " + x);
    return;
}
```

Here the `return` statement doesn't contribute to program execution or readability (we already know that the method returns when it hits bottom), but later we will see cases where explicitly saying `return` can be useful.

A call to a method with a non-void return type can be used anywhere that a variable or literal of the same type can be used: as the right-hand side of an assignment, as an operand in an operation, or even as an argument of another method call. By contrast, a call to a void method has no type. So if method `iAmVoid` is void, you could not say `int z = iAmVoid();` because there would be no value to assign to `z`. When you call a void method, you just want it to do its thing, so you make the call all by itself, followed by a semicolon, like this:

```
iAmVoid();
```

If a method changes the state of the program or the computer in any way (other than returning the return value), the change is called a *side effect*. Clearly, when you call a void method, you do so because you are interested in a side effect. (In this example, the side effect was the printing of the message.) Sometimes you might want to call a non-void method, not because you are interested in the return value, but because you want the side effect. In that case, you can just call the method as if it were void.

For example, the following method both prints out and returns hypotenuse squared:

```
static int vocalHypotSquared(int a, int b)
{
    int hSquared = a*a + b*b;
    System.out.println("h-squared = " + hSquared);
    return hSquared;
}
```

If you just wanted to print out the message, you could call the method like this, ignoring the return value:

```
vocalHypotSquared(5, 12);
```

Polymorphism

Polymorphism comes from the Greek for "many forms." It is one of several five-syllable words pertaining to object-oriented programming. In our context, it means that a method can have one name but many forms. In other words, you can define multiple methods with the same name.

At first, this might seem impossible. How can the system know which of the various methods you had in mind? The rule is that if two methods have the same name, their argument lists have to be different. That is, the types that appear in lists must differ; the argument names are not considered here. If a method name appears more than once, we say that the name is *overloaded*.

For example, the following two methods could appear in the same program:

```
static int getMass(int n)
{
    ...
}
static int getMass(double a, char c)
{
    ...
}
```

Here, `getMass()` is legitimately overloaded. However, the following two methods could *not* appear in the same program:

```
static int getMass(int n)
{
    ...
}
static int getMass(int x)
{
    ...
}
```

The argument names are different, but that doesn't help. Both method versions have the same name, and each version takes a single argument of type `int`, so the compilation will fail. If the argument types were different (for example, if the argument of `getMass()` had type `long` or `byte`), the code would compile without error.

Methods That Call Methods

Methods can be called from anywhere – even from other methods. In fact, any complicated program is likely to consist of methods that call other methods that call other methods, and so on, to many levels of depth. For example, you might have a method that prints out two values:

```
static void print2Vals(int val0, int val1)
{
    System.out.println(val0 + " and " + val1);
}
```


Now what if you want a method that prints out the cubes of its two arguments? You might do it as follows:

```
static void print2Cubes(int val0, int val1)
{
    int val0Cubed = val0*val0*val0;
    int val1Cubed = val1*val1*val1;
    System.out.println(val0Cubed + " and " + val1Cubed);
}
```

Since you already have the `print2Vals` method, you can rewrite `print2Cubes` as follows:

```
static void print2Cubes(int val0, int val1)
{
    int val0Cubed = val0*val0*val0;
    int val1Cubed = val1*val1*val1;
    print2Vals(val0Cubed, val1Cubed);
}
```

Now you have a method, (`print2Cubes`) that calls another method (`print2Vals`). You can rewrite `print2Cubes` to be even more terse, as follows:

```
static void print2Cubes(int val0, int val1)
{
    print2Vals(val0*val0*val0, val1*val1*val1);
}
```

Since the multiplication is repetitious, you can also introduce a new method:

```
static int nCubed(int n)
{
    return n*n*n;
}
```

Now `print2Cubes` becomes

```
static void print2Cubes(int val0, int val1)
{
    print2Vals(nCubed(val0), nCubed(val1));
}
```

Now you have a method, (`print2Cubes`) that consists of a single call to another method (`print2Vals`), and that call's arguments are both expressed as method calls (to `nCubed`). This kind of structure—calls to calls to calls—is perfectly typical of programs.

Passing by Value

In formal computer terminology, we say that Java passes *by value*. This is just another way to say that methods get copies of their arguments and have no access to the original values. The alternative is called passing *by reference*, where methods work with the caller's data and not with copies. The latter is a perfectly valid way to design a language; it's just not the way Java does it.

When a caller calls a method and the flow detours into the method's body, the JVM copies the argument values provided by the caller and gives the copies to the method. This means that if the method alters its arguments, the alteration has no effect on the caller.

Consider the following method:

```
static void print3x(int x)
{
    x = 3*x;
    System.out.println("3 times x = " + x);
}
```

The method triples its argument. You might wonder what happens to the value that the caller passes to the method. For example, what does the following print out?

```
int z = 10;
print3x(z);
System.out.println("Now z is " + z);
```

Does this code print "Now z is 10" or "Now z is 30"? If the method has access to the actual data passed in by the caller, the code should print out "Now z is 30". However, the code actually prints "Now z is 10" because the method triples its own private copy, not touching the

caller's version. After the method returns, the memory used for storing the method's copy is recycled. The method's copy does not survive after the method returns.

Order

A program is, in large part, a collection of methods that call other methods. These other methods call still other methods, and so on.

Within an application, methods can appear in any order. Once again, you are in a situation where your choices can make a program either easier or harder for others to read. It makes sense to put related methods near one another. It is common practice, though by no means universal, to put the `main` method at the very end, just before the final closing curly bracket. From here on, this book will follow that convention.

Scope

When you write a method declaration, you can choose almost any argument names you like. Of course, the names have to be legal (beginning with a letter, underscore, or dollar sign, and continuing with the same plus digits). Moreover, the name should be indicative of the argument's meaning. But beyond these considerations, you have complete latitude. In particular, you are allowed to reuse a variable name that has been used elsewhere in your program.

Every Java variable has a *scope*. A variable's scope is the matching pair of open and closed curly brackets that most tightly encloses the variable's declaration. Another way to say this is in terms of blocks. A *block* is a contiguous piece of code that begins with an open curly and ends with a matching closed curly. Blocks may contain many kinds of code. We have already seen blocks that contain method bodies. Later in this book, we will see blocks that contain, among other things, other blocks. Those inner blocks can contain, among other things, still other blocks, and so on, to whatever depth is useful.

Already we have seen blocks that contain other blocks, since every Java application is a block that looks like this:

```
1. public class ClassName
2. {
3.     // Optional other methods.
4.     public static void main(String[] args)
5.     {
6.         ...
7.     }
8. }
```

Any variable defined in the main method has a scope that spans from line 5 through line 7, since that is the tightest matched pair of curlys that would contain the variable's declaration.

The scope of a method argument is the method itself, even though the argument is actually declared just before the open curly that begins the scope.

Now, here is why it is so important to know about scope: A variable name may not be declared more than once in a single scope. However, a name that is declared in one scope may be declared and used in any number of other scopes. Each declaration refers to a different variable; the variables just happen to have the same name. The situation is similar to filenames in directories. Names must be unique within any particular directory, but a filename that appears in one directory may be used in another directory. The two files have nothing to do with each other, and the common name is just a coincidence.

Consider the following example:

```
1. public class ReusesNames
2. {
3.     static void printTriple(int x)
4.     {
5.         int i = 3*x;
6.         System.out.println("Triple = " + i);
7.     }
8.
9.     public static void main(String[] args)
10.    {
11.        int x = 10;
12.        int i = x+5;
13.        printTriple(i);
14.    }
15. }
```

Here we have two methods, `main` and `printTriple`, each with its own scope. Each method's scope has its own `i` and its own `x`, unrelated to the `i` and `x` of the other method. Each method can use and modify its own `i` and `x`, but cannot touch the `i` and `x` of the other method.

A convenient effect of Java's scoping rule is that when you write a method, you don't have to worry about whether a variable name you like is already in use in a different method. This is especially convenient in a long program that might have hundreds of methods, each with a dozen variables. If it were not for the scoping rule, we would quickly run out of good variable names.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Which of the following are legal method names?

1. \$25
2. 25\$
3. abc_
4. _ABc

2. Suppose you want to write a method that returns the diameter of a planet, in millimeters. Since it's your program, you can choose any name you like for the method. Rank the following method names, from worst to best. Use your own judgment as to what makes one method name better or worse than another.

1. getPlanetDiameter
2. getSize
3. getPlanetDiameterMm
4. getIt
5. getPlanetSize

3. Suppose a method has the following declaration:

```
static int abc(int x, short y)
```

Suppose this method is called as follows:

```
abc(first, second)
```

Which of the following are legal types for the variables `first` and `second`?

1. `int first, int second`
2. `short first, short second`
3. `byte first, char second`
4. `char first, byte second`

4. Consider the following method declaration:

```
xyz(double d)
```

Which argument types can a caller pass into this method?

5. Earlier in this chapter, you learned that if method `iAmVoid` is void, you can't say `int z = iAmVoid();` because there is no value to assign to `z`. What happens if you try? Write a program that does this experiment.

6. Earlier in this chapter, you saw the following method:

```
static void print3x(int x)
{
    x = 3*x;
    System.out.println("3 times x = " + x);
}
```

The following code prints out "Now z is 10", not "Now z is 30", because the method modifies its own private copy of the argument:

```
int z = 10;
print3x(z);
System.out.println("Now z is " + z);
```

Write a program that proves this.

Chapter 5: Conditionals and Loops

The [previous chapter](#) showed you how method calls can be used to detour the flow of program execution. This chapter will introduce two more ways to vary program flow: conditionals and loops. By the end of this chapter, you will be able to write programs in which control flows in quite intricate patterns.

Conditionals

If a method call is like a detour in the path of execution, then a *conditional* is like a fork in the road. Conditional code is executed only if a certain criterion is met, typically when a certain boolean expression evaluates to `true`.

We will begin with the `if` statement, which is Java's most basic conditional. We will also look at the more complicated ternary operator and `switch` statement.

if

In its simplest form, the `if` statement looks like this:

```
if (boolean_expression)
    do_something;
```

The code immediately following the `if` keyword must be of boolean type and must be enclosed in parentheses. The code that follows the parenthetical boolean expression can be either a single statement or a block of statements enclosed in curly brackets. Let's look at some examples.

The following code fragment prints out a message if `x` is divisible by 10:

```
if (x%10 == 0)
    System.out.println("x is divisible by 10.");
```

In the next example, `y` and `z` are both reduced if their product exceeds 1,000:

```
if (y*z > 1000)
{
    y -= 10;
    z -= 20;
}
```

Note in the previous example that if the condition is met, the action to be taken consists of two statements. When the conditional action is longer than a single statement, the multiple statements of the action are enclosed in curly brackets.

if and else

An `if` statement can be enhanced with the `else` keyword. You can only use `else` after the statement or curly bracket-enclosed block that follows an `if`. As with `if`, the code that follows `else` can be either a single statement or a block of statements within curly brackets. As you might expect, the code following `else` is executed if the `if` statement's boolean expression evaluates to `false`.

For example, the following code prints out a message that depends on whether the value of `x` is even or odd:

```
if (x%2 == 0)
    System.out.println(x + " is even.");
else
    System.out.println(x + " is odd.");
```

In the next example, the method "clamps" the value of its `z` argument. The return value is `z`, unless `z` exceeds a lower or upper limit. If this is the case, the return value is the exceeded limit:

```
static long clamp(long z, long lowLimit, long highLimit)
{
    if (z < lowLimit)
        return lowLimit;
    else if (z > highLimit)
        return highLimit;
    return z;
}
```

If curly bracket-enclosed code blocks are used after `if` or `else`, those blocks themselves can contain `if` statements. The following code fragment uses *nested* `if` statements:

```
if (x > 1000000)
{
    // x is big.
    if (x%2 == 0)
        System.out.println("Big and even.");
    else
        System.out.println("Big and odd.");
}
else
{
    // x is little.
    if (x%2 == 0)
        System.out.println("Small and even.");
}
```

```
else
    System.out.println("Small and odd.");
}
```

Note There is no limit to how deeply you can nest `if` statements. Of course, if you nest too deeply, your code becomes difficult to read and understand. See the "[Nesting](#)" section later in this chapter for an explanation of this technique.

else if

In the [previous section](#), you learned how to follow an `if` statement with an `else` statement. You can also follow an `if` statement with an arbitrary number of `else if` statements. An `else if` statement is like an `else` statement, but it is followed by a parenthetical boolean expression and then by a single statement or curly bracket-enclosed block. As you might expect, the single statement or curly bracket-enclosed block is executed only if the boolean expression evaluates to `true`. There is no limit to the number of `else if` statements that may follow an `if` statement, and the last `else if` statement may be followed by an `else` statement.

The following example is a method that prints out one of a number of possible messages, based on the size of the `z` argument:

```
static void howBig(double z)
{
    if (z < 0.001)
        System.out.println("Very tiny");
    else if (z < 1)
        System.out.println("Tiny");
    else if (z < 100)
        System.out.println("Medium");
    else if (z < 100000)
        System.out.println("Large");
    else
        System.out.println("Very large");
}
```

Note that the series of tests on the value of `z` begins with a straightforward `if` statement, followed by three `else if` statements. The `else` statement comes at the end, which is the only place where it may appear.

Before we continue, let's take a moment to appreciate the power of the various versions of the `if` statement. The Java functionality presented in the previous chapters of this book, while impressive, amounts to using your computer as a very fast calculator. For instance, a method would always process its arguments in exactly the same way. With the introduction of `if` statements, we have programs that can react flexibly. The `howBig` method, for example, can react flexibly to the value of its `z` argument.

Later in this chapter we will examine loops, which introduce an additional level of flexibility. But first, let's look at two more kinds of conditional execution: the ternary operator and the `switch` statement.

The Ternary Operator

In [Chapter 3, "Operations,"](#) we looked at Java's unary and binary operators. Now let's look at the *ternary* operator. The name *ternary* just means that there are three operands. Since there are three, we will need two symbols to separate them: the question mark (?) and the colon (:). The operator is used like this:

```
boolean_expression ? value_1 : value_2
```

The value of the ternary operation depends on the value of the boolean expression. If the boolean expression evaluates to `true`, the value of the overall operation is `value_1`. If the boolean expression evaluates to `false`, the value of the overall operation is `value_2`.

Typically, a ternary operation appears on the right-hand side of an assignment. For example, suppose you want `radius` to be 10 if `mass` is less than or equal to 50,000; otherwise, you want `radius` to be 99. Without the ternary operator, you could do it this way:

```
if (mass <= 50000)
    radius = 10;
else
    radius = 99;
```

You can rewrite this in a single line with the ternary operator:

```
radius = mass <= 50000 ? 10 : 99;
```

The boolean expression does not need to appear in parentheses, but the line is more readable like this:

```
radius = (mass <= 50000) ? 10 : 99;
```

The ternary operator is a convenient replacement for an `if...else` expression.

Now let's look at the `switch` statement, which is a convenient replacement for a sequence of `if...else` expressions.

switch

In the [previous section](#), you saw how the ternary operator can replace certain `if...else` structures. We will now look at the `switch` statement, which can replace entire chains of `if...else if...else if` structures.

Suppose you wanted to write some code that takes special actions if the value of a `char` called `theChar` is a vowel (a, e, i, o, or u). The special actions consist of printing a message and setting the value of an `int` called `vowelNum`. Using `if` and `else`, you could write the code as follows:

```
if (theChar == 'a')
{
    System.out.println("a is a vowel.");
    vowelNum = 0;
}
else if (theChar == 'e')
{
    System.out.println("e is a vowel.");
    vowelNum = 1;
}
else if (theChar == 'i')
{
    System.out.println("i is a vowel.");
    vowelNum = 2;
}
else if (theChar == 'o')
{
    System.out.println("o is a vowel.");
    vowelNum = 3;
}
else if (theChar == 'u')
{
    System.out.println("u is a vowel.");
    vowelNum = 4;
}
}
```

This can be rewritten as follows, using a `switch` statement:

```
switch (theChar)
{
    case 'a':
        System.out.println("a is a vowel.");
        vowelNum = 0;
        break;
    case 'e':
        System.out.println("e is a vowel.");
        vowelNum = 1;
        break;
    case 'i':
        System.out.println("i is a vowel.");
        vowelNum = 2;
        break;
    case 'o':
        System.out.println("o is a vowel.");
        vowelNum = 3;
        break;
    case 'u':
        System.out.println("u is a vowel.");
        vowelNum = 4;
        break;
}
```

The value in parentheses just after the `switch` keyword is called the *expression* of the `switch` statement, and it must be of type `byte`, `short`, `char`, or `int`. (This example assumes that `theChar` has been declared to be a `char`.) When the `switch` code is executed, Java searches through the `case` statements, looking for one that matches the expression's value. If no match is found, nothing happens; execution continues after the closing curly bracket. If a match is found, control jumps to the first executable line following the `case` statement. Then execution proceeds line by line until a `break` statement is reached. At this point, execution of the `switch` code is terminated, and control continues after the closing curly bracket.

switch and default

The keyword `default`, followed by a colon, can appear in place of a `case` statement. The code following the `default` statement is executed if none of the `case` statements match the expression. For example, suppose you want to modify your code so that it prints out "Not a vowel" if `theChar` is not a vowel. If you couldn't use a `switch` statement, you would do the following:

```
if (theChar == 'a')
{
    System.out.println("a is a vowel.");
    vowelNum = 0;
}
else if (theChar == 'e')
{
    System.out.println("e is a vowel.");
    vowelNum = 1;
}
else if (theChar == 'i')
{
    System.out.println("i is a vowel.");
    vowelNum = 2;
}
else if (theChar == 'o')
{
    System.out.println("o is a vowel.");
    vowelNum = 3;
}
```

```
}
else if (theChar == 'u')
{
    System.out.println("u is a vowel.");
    vowelNum = 4;
}
else
    System.out.println("Not a vowel.");
```

This code is the same as the original solution, but with a final `else` at the end. The following code uses a `switch` statement with a default block to achieve the same result:

```
switch (theChar)
{
    case 'a':
        System.out.println("a is a vowel.");
        vowelNum = 0;
        break;
    case 'e':
        System.out.println("e is a vowel.");
        vowelNum = 1;
        break;
    case 'i':
        System.out.println("i is a vowel.");
        vowelNum = 2;
        break;
    case 'o':
        System.out.println("o is a vowel.");
        vowelNum = 3;
        break;
    case 'u':
        System.out.println("u is a vowel.");
        vowelNum = 4;
        break;
    default:
        System.out.println("Not a vowel.");
        break;
}
```

When you look at all four versions of this example, you can see that using a `switch` statement does not significantly reduce the number of lines of code (although there is a reduction). The main benefit is readability. The `switch` versions more clearly tell readers what is happening.

Omitting the *break*

Once a `case` block is found that matches the `switch` statement's expression, execution continues until a `break` is reached or the `switch` statement's closing curly bracket is reached, whichever comes first. If a `case` block does not end with a `break`, execution continues past the next `case` statement and into the code for that `case` block.

In the previous example, suppose the `case` block for 'e' did not end with a `break` statement. (Perhaps due to an innocent oversight. It's only human to forget to type `break` from time to time.) The code would then look like this:

```
. . .
7. case 'e':
8.     System.out.println("e is a vowel.");
9.     vowelNum = 1;
10. case 'i':
11.     System.out.println("i is a vowel.");
12.     vowelNum = 2;
13.     break;
. . .
```

We've added line numbers for easy reference. The `switch` statement detects that the expression value ('e') matches the case on line 7. The message on line 8 is printed out, and then at line 9 `vowelNum` is set to 1. Since there is no `break` at line 10, execution just keeps on going. The `case` statement at line 10 is ignored, and control flow continues at line 11. The message on line 11 is printed out, and at line 12 `vowelNum` is set to 2. At last we have a `break`, so execution of the `switch` is finished.

This behavior of continuing from one case to the next in the absence of a `break` statement is called *falling through*. Falling through is a mixed blessing. When it happens because you forgot to type `break` for a particular case, it's just a bug that might be hard to find (but easy to fix once it's found).

On the other hand, falling through might be just the behavior that you want. The feature is especially useful when you want to use the same code to process more than one case. The letters `y` and `w` are sometimes considered to be vowels. (*Y* occasionally, as in *occasionally*; *w* very rarely, as in *crwth*, a medieval musical instrument, pronounced "crooth.") You might want to print out a special message if `theChar` has either of these values. If you were using `if...else` code, you would insert the following lines:

```
. . .
else if (theChar == 'y' || theChar == 'w')
    System.out.println("y and w are sometimes vowels.");
. . .
```

You can incorporate this test into your `switch` code by inserting the following lines:

```
. . .
case 'y':
case 'w':
    System.out.println("y and w are sometimes vowels.");
    break;
. . .
```

Where should these lines be inserted? Strictly speaking, the cases in a `switch` statement, including the `default` code, can appear in any order. However, for readability, it makes the most sense to have the cases appear in their natural order (numerical or alphabetical), with the `default` code appearing last.

Now that we have looked at Java's conditional code, we can turn our attention to loops.

Team LiB

← PREVIOUS NEXT →

Loops

You have seen that conditional code is like a fork in the path of program execution. Extending this analogy, a loop is like an eddy or a whirlpool. No, wait, that can't be right... paths don't have whirlpools. The analogy has broken down. At any rate, a loop is a piece of code that is executed repeatedly. The number of repetitions can be some preset value, or the loop can run on and on until a condition is met.

We will begin with `while` loops, and then move on to `for` loops. We will also look at several techniques for enhancing loop behavior: breaking, continuing, and nesting.

While Loops

A while loop is a chunk of code that is executed repeatedly until a certain condition is met. The format for a while loop is

```
while (expression)
    loop_body
```

The expression must be of the boolean type. The loop body is the code to be repeated. This can be either a single statement or a block of code enclosed in curly brackets. Initially the expression is evaluated, and if its value is `true`, the loop body is executed once. Then the expression is evaluated again, and if its value is still `true`, the loop body is executed once again. This happens again and again and again. Eventually (we hope), the expression evaluates to `false`. When this happens, the loop body is not executed anymore. Instead, control jumps to the code immediately after the loop body.

This explanation might seem paradoxical. If the while loop is to be of any use, the expression must initially evaluate to `true`. Otherwise, the loop body won't be executed at all. But if the expression is indeed initially `true`, how can the loop ever terminate?

The answer is that either the expression or the loop body must modify the data from which the expression is calculated. Let's look at a few examples of how this works.

First, here is a useless loop that prints too many messages:

```
int x = 23;
while (x > 0)
    System.out.println("Still going!");
```

This loop runs forever or until you press Ctrl+C to terminate the program, whichever comes first. This example demonstrates the need to somehow modify the data that constitutes the expression.

The next example is more useful. The following code prints the numbers 1 through 10:

```
int counter = 1;
while (counter <= 10)
{
    System.out.println("counter = " + counter);
    counter += 1;
}
```

You can use a pre-increment or post-increment operator to make this code slightly more terse:

```
int counter = 1;
while (counter <= 10)
{
    System.out.println("counter = " + counter);
    counter++;
}
```

The next example prints out consecutive square numbers that are less than 1,000:

```
int counter = 1;
while (counter*counter < 1000)
{
    System.out.println(counter * counter);
    counter++;
}
```

This example points out a useful feature of while loops: You don't need to know beforehand how many passes you want to make through the loop body. You could certainly find a calculator, figure out that the square root of 1,000 is 31.6227766..., and write the following:

```
int counter = 1;
while (counter <= 31)
{
    System.out.println(counter * counter);
    counter++;
}
```

This approach works, but it violates the spirit of making the computer compute. If you're programming a computer, you shouldn't have to reach for a calculator. With the next-to-last version of the example, you take advantage of the fact that you don't have to know how many passes you're going to make through a while loop. You just have to be able to know when you're done.

While loops are the first of several kinds of loops that we will present in this chapter. Loops are powerful because a few lines of source code can cause the computer to execute a very large number of instructions.

The WhileLab animated illustration demonstrates while loops. To run the program, type `java loops.WhileLab`. You see a display that shows a while loop with two assignment lines in its body, as shown in [Figure 5.1](#).

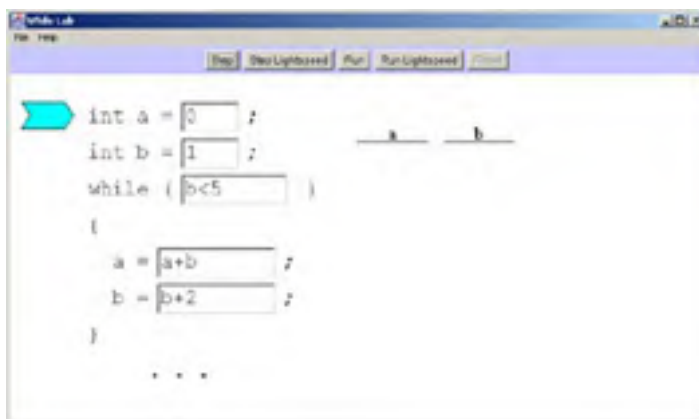


Figure 5.1: While Lab: initial display

The loop uses variables `a` and `b`. As the code executes, their values are displayed and updated. Click on the Step button to animate the next line of code. Click on the Run button to animate the entire loop. You can click on Step Lightspeed or Run Lightspeed to bypass the animation and just see the result. When the animation is finished, click Reset to start again.

You can type in your own values for the initial values of `a` and `b`, for the test expression, and for the new values that are assigned to `a` and `b` within the loop. Figure 5.2 shows While Lab with a slightly modified test expression.



Figure 5.2: While Lab with modified test expression

Figure 5.3 shows the result of executing the configuration shown in Figure 5.2.

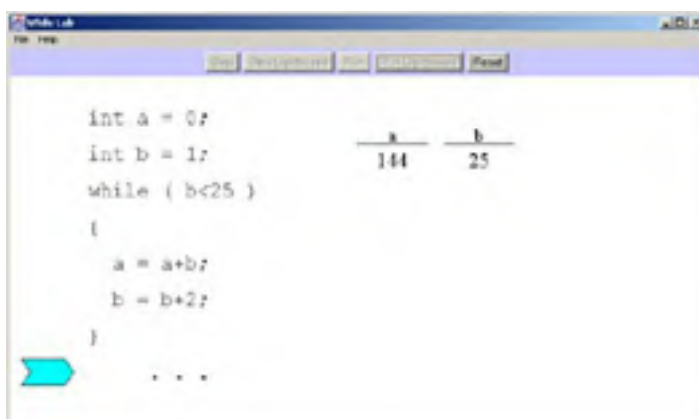


Figure 5.3: While Lab after execution

Try typing in different values for `b <= ??` in the test expression. (If you enter a large number, the loop will be executed a large number of times, so you probably want to execute with the Run Lightspeed button rather than the Run button.) As you vary the limit on `b`, what do you notice about the final value of `a`?

Try configuring WhileLab's code display so that the code computes the following results (which can be in either `a` or `b`, whichever you prefer):

- The sum of the numbers 1 through 500, inclusive.
- The sum of the even numbers from 50 through 60, inclusive.
- The product of the first five odd numbers.

Now that you have some experience with while loops, we can look at a variation on the theme: do-while loops.

Do-While Loops

A while loop always tests its condition before executing its body. There may be times when you want to execute the body first, and then test. This is done with a do-while loop. The format of a do-while loop is

```
do
    loop_body
while (expression);
```

As with ordinary while loops, the loop body can be either a single statement or a curly bracket-enclosed block. Note that the parenthetical expression must be followed by a semicolon.

When a do-while loop is executed, the loop body is executed. Then the expression is evaluated. If the expression evaluates to `true`, the loop body is executed again, the expression is evaluated again, and so on until eventually the expression value is `false`. At that point, execution of the loop is finished. As with ordinary while loops, you should write do-while code in such a way that during execution of the loop, the data constituting the expression changes so that at some point the expression's value can become `false`.

The code in the following example prints out the cube of `x`, and then increments `x` by 5, until `x` exceeds 100:

```
do
{
    System.out.println(x*x*x);
    x += 5;
}
while (x <= 100);
```

The body of a do-while loop is always executed at least once. In the preceding example, at least one line will be printed out, even if the initial value of `x` is greater than 100.

Do-while loops are not very different from while loops. The main difference is that the body of a while loop might not ever be executed, whereas the body of a do-while loop will always be executed at least once.

Now let's look at for loops, which are useful when you know how many passes through the loop body you want.

For Loops

The following code, which uses a while loop to compute a value and print a message ten times, has a very common structure:

```
int z = 0;
while (z < 10)
{
    int formula = z*z*z + z*z;
    System.out.println(formula);
    z++;
}
```

The code first initializes `z`, and then it enters a while loop. Within the loop body, the first two lines perform the internal business of the loop, so to speak. The last line (`z++`) is concerned with updating the only data that changes from pass to pass in the loop. [Figure 5.4](#) shows the structure of the loop.

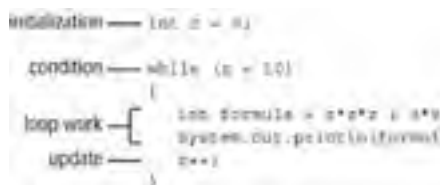


Figure 5.4: A common loop usage

This structure (initializing before a loop, incrementing at the end of the loop body) is so common that there is a special kind of loop to support it. The `for` loop has the following format:

```
for (initialization; condition; update)
    body
```

This `for` loop is exactly equivalent to

```
initialization;
while (condition)
{
    body
    update
}
```

The `for` keyword must be followed by three items, known as the *initialization*, the *condition*, and the *update*. These are enclosed in parentheses and separated by semicolons. When the `for` loop is processed, the initialization is executed once. Then the condition, whose type must be boolean, is evaluated. If the condition is `true`, the loop body is executed. Then the condition is evaluated again, and so on until the condition is `false`. Notice that no matter how many times the loop body is executed (zero times, once, or multiple times) the initialization is executed exactly once.

`For` loops are useful when you know beforehand how many times you want the loop body to be executed. (They are especially useful when you're processing arrays, which will be presented in the [next chapter](#).) When you don't know beforehand how many times the body should execute, you are generally better off using a while loop because your code will be less complicated.

Usually, the initialization involves setting the value of a single variable, often to zero. The condition is usually a test on the value of that variable, and the update increments the variable. For example, the following code prints out a message 10 times:

```
int i;
for (i=0; i<10; i++)
    System.out.println("DANGER!");
```

In this code, the variable `i` is used just to regulate the number of passes through the loop body. Since it does not appear in the body, we could have chosen any name for the variable, but it is conventional to use `i` (or `j` if `i` is in use). A variable used in this way (regulating the number of passes through the loop body, but otherwise playing little or no role in the body) is called a *loop counter*. We could have initialized `i` to any value, as long as the value in the condition was 10 greater than that, but it is conventional to start a loop counter at 0. If you follow these conventions, and we strongly recommend that you do so, people who read your code will have a good chance of understanding your intentions.

The initialization and update portions of a for loop can have multiple parts, separated by commas. For example, the following code prints out the areas of rectangles whose bases range from 5 to 10 inches, and whose heights are 2 inches more than the base:

```
int base, height;
for (base=5, height=7; base<=10; base++, height++)
{
    int area = base * height;
    System.out.println(area + " square inches");
}
```

Here, both the initialization and the update have multiple parts.

Breaking and Continuing

Usually a loop runs until its condition is `false`. However, there may be times when you want to terminate the loop prematurely. This is called *breaking out of the loop*.

As an example of loop breaking, imagine you are writing a payroll program for a small company. The company has 100 employees whose ID numbers are 1001 through 1100. A method called `getPayAmount`, which takes an employee ID as its argument, returns the amount of money the employee should be paid. Another method called `printCheck`, which has an employee ID and an amount as its arguments, prints the specified employee's paycheck. A variable called `balance` keeps track of how much money the company has in the bank.

The following code prints everybody's paycheck and keeps track of the bank balance:

```
int id;
for (id=1001; id<=1100; id++)
{
    float pay = getPayAmount(id);
    printCheck(id, pay);
    balance -= pay;
}
```

The problem with this code is that it doesn't take precautions against using up all the money in the bank account. The following code uses a `break` statement to terminate the loop as soon as there isn't enough money left to cover the next paycheck:

```
int id;
for (id=1001; id<=1100; id++)
{
    float pay = getPayAmount(id);
    if (balance-pay < 0)
        break;
    printCheck(id, pay);
    balance -= pay;
}
```

The `break` statement causes immediate termination of the loop. Execution continues with the first line of code following the loop. You can break out of any kind of loop: `do`, `do-while`, and `for`. Note that this application of `break` is unrelated to using `break` to terminate a case in a `switch` block. The two situations are very different.

There might be times when you want to terminate not the entire loop, but just the current pass through the loop. You do this with the `continue` statement. The following example uses `continue` in a loop that prints out the square and cube of every number from 1 through 20, except 8:

```
int i;
for (i=1; i<=20; i++)
{
    if (i == 8)
        continue;
    int squared = i * i;
    int cubed = squared * i;
    System.out.println(squared + ", " + cubed);
}
```

The `continue` statement causes control to jump to the end of the loop body. Then the update (`i++`) is executed, the condition is checked, and perhaps more passes are made through the loop body. In other words, the current pass through the loop body is terminated prematurely. As with `break` statements, you can use `continue` statements with `do` and `do-while` loops as well as with `for` loops.

The `continue` statement allows you to improve on the preceding paycheck example, which broke out of the loop as soon as you couldn't afford to pay a salary. This was possibly unfair to the workers who had not yet been paid. After all, the employee who caused the break might have had the highest salary in the company. Even if there was not enough money to pay that person, there might still be enough left to pay someone else. So a more fair version of the program would be

```
int id;
for (id=1001; id<=1100; id++)
{
    float pay = getPayAmount(id);
    if (balance-pay < 0)
        continue;
    printCheck(id, pay);
    balance -= pay;
}
```

The only difference between this version and the previous one is the replacement of `break` with `continue`. Now the loop never terminates prematurely, although some passes through loop body might do so.

Nesting

The body of a loop can contain any valid Java code, including another loop, which can itself contain any valid Java code, including another loop, and so on. The technique of putting a loop within a loop is called *nesting*.

As an example of loop nesting, suppose you are writing code to generate frames for an animated movie. Assume that a frame consists of a grid of 1000 x 1000 pixels. (*Pixel* is an abbreviation for *picture element*. A pixel is a tiny dot of color, almost too small to see. If you hold a magnifying glass up to your computer screen, you can see the individual pixels.) Assume also that there is a method called `computePixel`, which takes as arguments the horizontal and vertical positions of the pixel whose color value is to be computed. Fortunately, `computePixel` also stores the color value in the appropriate place, so all you have to worry about here is calling the method with the right arguments.

The following code uses nested for loops to call `computePixel` for every pixel position:

```
1. int x, y; // x = horiz, y = vert
2. for (y=0; y<1000; y++)
3.   for (x=0; x<1000; x++)
4.     computePixel(x, y);
```

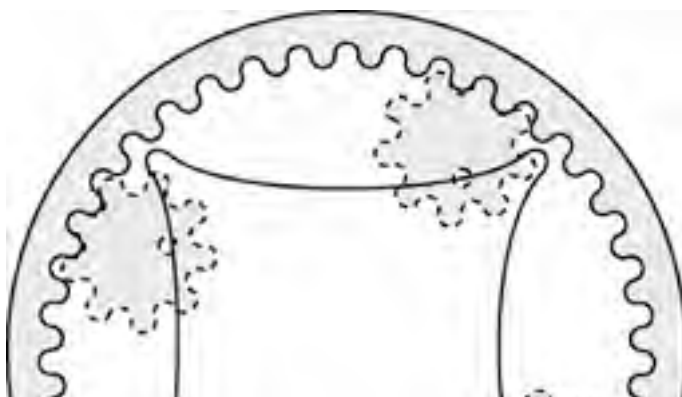
It is conventional to use `x` as a variable name for representing horizontal positions, and `y` for representing vertical positions. Line 2 says that the outer loop body will be executed with `y` ranging from 0 through 999. The outer loop body is lines 3 and 4. Line 3 says that the inner loop body, which is line 4, will be executed with `x` ranging from 0 through 999. So the first value pair passed to `computePixel` at line 4 will be (0, 0), followed by (1, 0), and then (2, 0), and then (3, 0), and so on up to (999, 0). Those thousand calls are the first pass through the outer loop. Then `y` is incremented from 0 to 1 and compared to 1,000. Since `y` is found to be still less than 1,000, the second pass through the outer loop begins: `computePixel` is called with arguments of (0, 1), (1, 1), through (999, 1). Every pass through the outer loop entails a thousand passes through the inner loop, until finally `computePixel` is called with arguments of (999, 999). At this point, the outer loop's condition is `false`, so the outer loop is finally done.

The body of the outer loop is two lines long (lines 3 and 4), but due to a technicality, the lines do not have to be enclosed in curly brackets. This is because, technically speaking, lines 3 and 4 are a single statement: a for loop. Only bodies consisting of multiple statements need to be enclosed in curly brackets. The precise definition of a statement is extremely intricate, but statements are easy to recognize because they end with semicolons. So you can get away with omitting curly brackets in this example, although you should indent responsibly to make it clear to readers that you are using a nested loop. However, it does no harm to add the curly brackets anyway (it's okay to have a block that contains just a single statement). The curly brackets do not affect execution speed, and they make the code a bit more readable, as you can see here:

```
1. int x, y; // x = horiz, y = vert
2. for (y=0; y<1000; y++)
3. {
4.   for (x=0; x<1000; x++)
5.   {
6.     computePixel(x, y);
7.   }
8. }
```

The `NestedLoopLab` animated illustration lets you use loops to draw cycloids. Cycloids are beautiful, complex geometric shapes. If you have ever played with a Spirograph, you have already appreciated cycloids. You could create some wonderful images with a Spirograph by drawing several curves in the same space but varying the curves' orientation or some other feature. If you have done this, you have performed a repetitive complicated task, varying features from one repetition to the next. In other words, you have done something that can be modeled with a loop, or possibly with nested loops.

A cycloid is the curve traced out by a point on a circle (the *roller*) as it rolls without slipping around the inside of a larger circle (the *gasket*). The roller always touches the gasket at one point, as shown in [Figure 5.5](#).



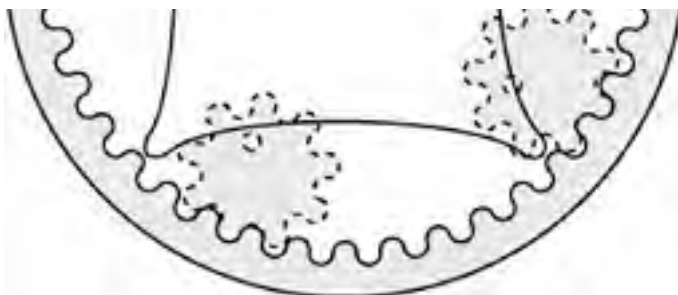


Figure 5.5: A cycloid

The ratio between the size of the roller and the size of the gasket determines the number of lobes the curve will have. The ratio in [Figure 5.5](#) is 1:4, so the curve has 4 lobes.

The *inset* is the distance from the tracing point to the rim of the roller. NestedLoopLab uses arbitrary inset units of 0 through 10, where 0 is the rim of the roller and 10 is the center. The orientation is the point of initial contact between the roller and the gasket, measured in clockwise degrees from straight up.

NestedLoopLab lets you select a ratio and color. You also choose a loop style. The default is no loop, but you can choose Loop to select a loop that varies the inset or the orientation. For really sophisticated images, you can use nested loops that vary the inset and the orientation, in either order. The Color choice lets you leave the color constant or vary the color in any of the loops

To launch NestedLoopLab, type `java loops.NestedLoopLab`. You will first see the display shown in [Figure 5.6](#).



Figure 5.6: NestedLoopLab: initial display

[Figure 5.7](#) shows NestedLoopLab with a ratio of 8:15 and a small inset.

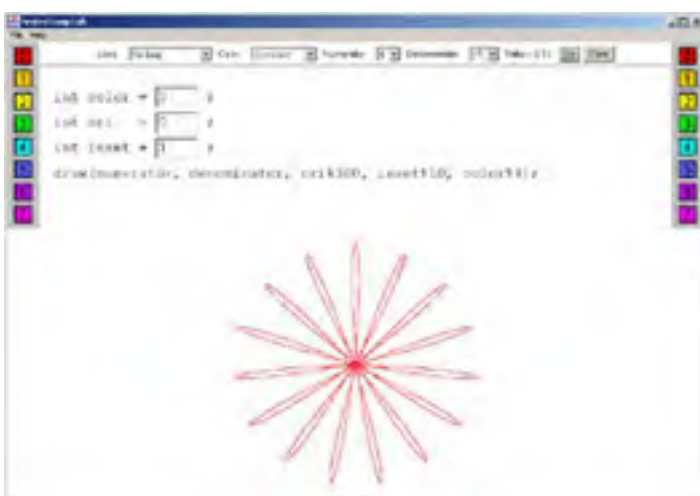


Figure 5.7: NestedLoopLab: 8:15

In [Figure 5.8](#), the configuration uses a loop, with the inset ranging from 0–6.

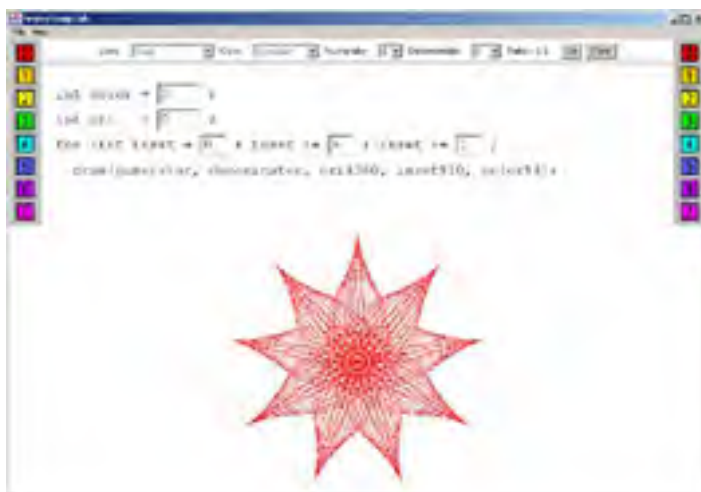


Figure 5.8: NestedLoopLab with a loop



Figure 5.9: NestedLoopLab with nested loops

Now try it for yourself. Enjoy! You can use the File , Gallery menu to view a few sample patterns. If you come up with a really spectacular image, please e-mail its settings to me at www.sybex.com so we can include it in future revisions of the gallery.

Labeled *break* and Labeled *continue*

Breaking out of a hierarchy of nested loops can be difficult. It might happen that code in an inner loop detects a condition that should cause an outer loop to be terminated. For example, you might use three nested loops to print paychecks for every employee in every department in every division of a large company. (Each department uses its own set of employee IDs, starting from zero.) The `getPayAmount` method now takes three arguments: division, department, and ID. Let's assume another feature for this method: The `if` statement will return a negative value if the corporate database that it consults is down. When this happens, paycheck processing should be terminated at once.

The following code would not be correct:

```
1. int  divn, dept, nDepartments, nEmployees, id;
2. float  pay;
3.
4. for (divn=0; divn <nDivisions; divn++)
5. {
6.     nDepartments = getDepartmentCount(divn);
7.     for (dept=0; dept <nDepartments; dept++)
8.     {
9.         nEmployees = getEmployeeCount(divn, dept);
10.        for (id=0; id<nEmployees; id++)
11.        {
12.            pay = getPayAmount(divn, dept, id);
13.            if (pay < 0)
14.                break;
15.            printCheck(divn, dept, id, pay);
16.            balance -= pay;
17.        }
18.    }
19. }
```

The code assumes the presence of methods that return the number of departments in a division (`getDepartmentCount`) and the number of employees in a department (`getEmployeeCount`). It also assumes that `nDivisions` has been preset to the

number of divisions in the company. The variable names `nDivisions`, `nDepartments`, and `nEmployees` mean, of course, the number of divisions, the number of departments, and the number of employees. This kind of naming convention is common and useful.

Unfortunately, the code doesn't work. The `break` keyword breaks out of the immediately enclosing loop, not out of all loops. So the `break` at line 14 just breaks out of the innermost loop (lines 10-17). Processing then continues with the next department because we are still in the middle loop (lines 7-18).

There is a simple but clumsy solution, which is shown in the following code. The innermost loop, when it detects a database problem, sets a boolean variable to `true`. The middle and outer loops have to check this variable and do their own break if it is `true`. Incidentally, a boolean variable that's used in this way to indicate program status is often called a *flag*. Here is the correct but clumsy code:

```
1. int    divn, dept, nDepartments, nEmployees, id;
2. float  pay;
3. boolean trouble = false;
4.
5. for (divn=0; divn <nDivisions; divn ++)
6. {
7.     nDepartments = getDepartmentCount(divn);
8.     for (dept=0; dept <nDepartments; dept ++)
9.     {
10.        nEmployees = getEmployeeCount(divn, dept);
11.        for (id=0; id<nEmployees; id++)
12.        {
13.            pay = getPayAmount(divn, dept, id);
14.            if (pay < 0)
15.            {
16.                trouble = true;
17.                break;
18.            }
19.            printCheck(divn, dept, id, pay);
20.            balance -= pay;
21.        } // End of inner loop
22.        if (trouble)
23.            break;
24.    } // End of middle loop
25.    if (trouble)
26.        break;
27. } // End of outer loop
```

The code is difficult to read, which is a good indicator of code that is needlessly complicated. The solution is Java's *labeled break* feature. This feature lets you assign a name, or *label*, to any `for`, `while`, or `do-while` loop. The label is any valid identifier (so you just need to follow the same rules that govern variable or method names). The label, followed by a colon, appears just before the `for`, `while`, or `do` keyword. Now you can break out of the labeled loop, even from code in a loop nested inside the labeled loop, by adding the loop's label after the `break` keyword.

The following code elegantly fixes the paycheck program by using a labeled loop and a labeled break:

```
1. int    divn, dept, nDepartments, nEmployees, id;
2. float  pay;
3.
4. bigloop: for (divn=0; divn <nDivisions; divn ++)
5. {
6.     nDepartments = getDepartmentCount(divn);
7.     for (dept=0; dept <nDepartments; dept ++)
8.     {
9.        nEmployees = getEmployeeCount(divn, dept);
10.        for (id=0; id<nEmployees; id++)
11.        {
12.            pay = getPayAmount(divn, dept, id);
13.            if (pay < 0)
14.                break bigloop;
15.            printCheck(divn, dept, id, pay);
16.            balance -= pay;
17.        }
18.    }
19. }
```

Now line 14 causes the outermost loop to break.

Java also provides a labeled `continue` feature. The statement `continue label;` terminates the current pass through the labeled loop. If the label were omitted, the current pass through the innermost loop would terminate instead.

Loops and Scope

The [previous chapter](#) introduced the concept of scope. As a reminder, a variable's scope is the block (inside curly brackets) that most tightly encloses the variable's declaration. *The variable has definition only within its scope.* Outside the scope, the variable's name may be reused, but the name refers to a different piece of data with its own scope.

With the introduction of loops, you begin to use code that can consist of blocks nested in blocks nested in blocks, and so on. This raises the issue of where variables should be declared. A good rule of thumb is that a variable's scope should be as small as possible. This means that if a variable is used only in a loop, it should be declared inside the loop.

For example, in the paycheck code of the [previous section](#), you declared `float pay` outside the outermost loop, even though it is only used in the innermost loop. A clear approach would be to eliminate the declaration on line 2 and change the innermost loop to the following:


```
for (id=0; id<nEmployees; id++)
{
    float pay = getPayAmount(divn, dept, id);
    if (pay < 0)
        break bigloop;
    printCheck(divn, dept, id, pay);
    balance -= pay;
}
```

Now anyone who reads the code and wonders where `pay` is used only has to think about five lines.

You are allowed to declare a variable in the initialization statement of a for loop. Thus, the following is allowed (and, in fact, is encouraged):

```
for (int i=0; i<10; i++)
{
    // Loop body
    // More loop body
}
```

The scope of `i` is the body of the loop.

Team LIB

PREVIOUS NEXT

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Rewrite the following code to maximize readability:

```
switch (x)
{
    case 100:
        System.out.println("x is big");
        break;
    case 101:
        System.out.println("x is big");
        break;
    case 10:
        System.out.println("x is medium");
        break;
    case -1000:
        System.out.println("x is negative");
        break;
}
```

2. Rewrite the following code to make it cleaner:

```
boolean flag = false;
switch (a)
{
    case 1:
        x = 1000;
        flag = true;
        break;
    case 30:
        y = 1000;
        flag = true;
        break;
}
if (!flag)
    z = 1000;
```

3. What happens when the following code is executed with `val` equal to 10? 100? 1,000? First, decide just by looking at the source code. Then write a program to verify your answer.

```
switch (val)
{
    case 10:
        System.out.println("ten");
    case 100:
        System.out.println("hundred");
    default:
        System.out.println("thousand");
}
```

4. Run the WhileLab animated illustration by typing `java loops.WhileLab`. Try changing the value in the condition in the third line. What do you notice about the final value of `a`?
5. The description of WhileLab suggests three exercises, which are repeated here. For each desired result, configure the inputs of WhileLab to produce that result. Then verify your work (and make sure WhileLab is trustworthy) by writing an application that duplicates each while loop. The loops should generate the following results:
 - The sum of the numbers 1 through 500, inclusive.
 - The sum of the even numbers from 50 through 60, inclusive.
 - The product of the first 5 odd numbers.
6. There is a number game called Hotpo that can entertain you for a few minutes while you're stuck in traffic, waiting for a movie to start, or having dinner with someone really boring. Hotpo stands for Half Or Triple Plus One, and it works like this: Think of an odd number. Now mentally calculate another number, as follows: If the first number was even, the next number is half the first one; if the first number was odd, the next number is 3 times the first number, plus 1. Now you can forget the first number and apply the Half Or Triple Plus One formula to your current number. Keep going until the value reaches 1. Let's try this with a starting number of 5. The series is $5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$.

Write a program that plays Hotpo. First, initialize a variable called `n` to the starting value you're interested in. Then enter a loop that prints out each number in the sequence, along with the current step number. For example, the output for 3 would be

```
Step #1: 10
Step #2: 5
Step #3: 16
Step #4: 8
Step #5: 4
Step #6: 2
Step #7: 1
```

Should the program use a while loop or a for loop?

7. What is the value of n after the following code is executed?

```
int n = 1;
outer: for (int i=2; i<10; i++)
{
    for (int j=1; j<i; j++)
    {
        n *= j;
        if (i*j == 10)
            break outer;
    }
}
```

Team LIB

PREVIOUS NEXT

Chapter 6: **Arrays**

Overview

All of the data types presented so far in this book have represented single units of information. The `char` type represents a single character, while the other types represent numbers with various formats and ranges.

This chapter will introduce arrays, which are clusters of data. You will learn how to create arrays and how to use them in programs, especially in the context of loops.

Arrays are extremely basic examples of objects. We can't claim that when you master arrays, you will have mastered object-oriented programming. However, in the course of this chapter, you will learn a number of concepts that will make it easy for you to master objects when they are presented in the next several chapters.

Clusters of Data

An *array* is a cluster of variables, called *components*, all of the same type. The array has a name, but the individual variables within the array do not. Each of the components has a unique identifying integer, called an *index*. The plural of index is *indices*, which proves that someone was paying attention in Latin class. The indices range from 0 through $n-1$, where n is the number of components in the array.

Before you use an array, you have to declare its type and create it. You have already seen type declarations in the context of primitive data types, and array declaration is quite similar. Creation is a new concept, and we will discuss it in some depth.

Declaring Arrays

Array declaration, like primitive declaration, associates a variable name with a data type. There are two ways to declare an array. The preferable format is

```
Component_type[] name;
```

Note the square brackets after the component type. An example of this format is

```
float[] temperaturesCelsius;
```

This declaration says that `temperaturesCelsius` is the name of an array whose components are all of type `float`. The number of components will be specified later, when the array is created. Note the plural in the name, indicating that the variable relates to more than one temperature.

The alternative format for array declaration is

```
Component_type name[];
```

The only difference is that the empty square brackets now come after the name, rather than after the component type. This format is included as a holdover from older programming languages (C and C++). It is considered less readable than the first format, and we will not use it in this book.

Creating Arrays

At some point after you declare an array, you need to create it. This is new. With primitives, all you had to do was declare a variable's type, and the variable came into existence. Arrays, as well as all other kinds of objects, are different: You have to create them explicitly. This is done with the keyword `new`.

The format of an array creation statement is

```
name = new component_type[number_of_components];
```

In the [previous section](#), you declared a variable named `temperaturesCelsius` to be an array whose components have `float` type. From this point on, we will say this more briefly: `temperaturesCelsius` is an array of floats. When you want to create the array, you first have to decide how many components it will have. If you want 10 components, for example, you would create the array like this:

```
temperaturesCelsius = new float[10];
```

The array size does not have to be a literal int. A variable is acceptable. Suppose you have a method called `getNTempReadings`, which returns the number of temperatures available to the program. Then you might do the following:

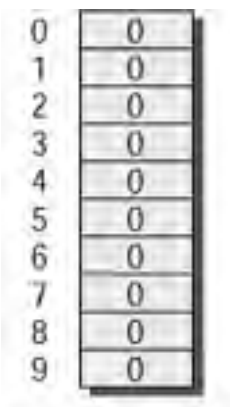
```
int nTemps = getNTempReadings();  
temperaturesCelsius = new float[nTemps];
```

You could even get more terse:

```
temperaturesCelsius = new float[getNTempReadings()];
```

When an array of primitives is created, all of its components are given an initial value. Numeric arrays (that is, arrays of `byte`, `short`, `int`, `long`, `float`, and `double`) have all components initialized to 0. Boolean arrays have all components initialized to `false`. Char arrays have all components initialized to the null character, which is a non-printing, do-nothing zero value that indicates "no character at all."

It is convenient to represent an array as shown in [Figure 6.1](#).



temperaturesCelsius

Figure 6.1: A new array

Figure 6.1 shows the array after it has been created but before any of its components have been modified. Now is the perfect time to learn how to modify the components.

Using Array Components

The components of an array of n components have indices from 0 through $n-1$. The names of those components are `arrayName[0]`, `arrayName[1]`, and so on, through `arrayName[n-1]`. Thus, in this example, you could use the following code to set the first and last components to -10 and 10, respectively:

```
temperaturesCelsius[0] = -10;  
temperaturesCelsius[9] = 10;
```

Sometimes the first component is called the *zeroth* component, so that the adjective will match the index. The term eliminates confusion, because one could reasonably (but wrongly) believe that the first component is `temperaturesCelsius[1]`.

Figure 6.2 shows the array after the preceding code has been executed.

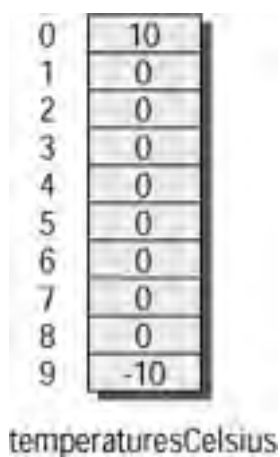


Figure 6.2: A used array

Of course, you can also read the value of an array component. The following code sets a component to the average of two other components:

```
temperaturesCelsius[5] =  
    (temperaturesCelsius[6] + temperaturesCelsius[7]) / 2;
```

You can see that an individual array component can be used in any context where a primitive variable of the same type can be used.

Array Length

When you create an array, you specify the number of components. Subsequently, the array knows its component count. You can read the number of components with the expression `arrayName.length`. For example, if you have an array called `employeeNames`, the following code sets an `int` called `nEmployees` to be the length of the array:

```
nEmployees = employeeNames.length;
```

The array's length is permanently fixed at creation time, so you can't modify it. The following code would cause a compilation error:

```
employeeNames.length = 5000;
```

Array Initialization

We have already said that when an array is created, its components are initialized to zero for numeric types, or to `false` or the null character for booleans and chars. If you want the array to start life with different contents, you can set the values of the components you want to change one by one, such as follows:

```
char[] chars = new char[10];  
chars[3] = 'L';  
chars[4] = 'C';
```

If you want to specify a value for all the components of the new array, a more compact syntax is available:

```
name = new type[] {value0, value1, ... };
```

The compiler determines the array length by counting the values in the curly brackets. The array components are initialized to those values. For example, the following code creates and initializes an array of 4 bytes:

```
byte[] bytes = new byte[] { 3, 5, 7, 99};
```


Arrays and Loops

Loops, and especially *for* loops, are ideal for processing arrays. No matter what you want to do to the array, you typically use a *for* loop with a loop counter that ranges from 0 through `array-length-minus-1`. Within the loop's body you perform whatever processing you want, using the loop counter as an array index.

For example, the following code computes the product of all the values in an array called `measurements`:

```
double product = 1;
for (int i=0; i<measurements.length; i++)
    product *= measurements[i];
```

Note that this code works on arrays of all sizes, because it reads the array size from `measurements.length`.

For another example, let's revisit the paycheck-printing code from the [previous chapter](#). Here is one of the several versions of that code:

```
int id;
for (id=1001; id<=1100; id++)
{
    float pay = getPayAmount(id);
    printCheck(id, pay);
    balance -= pay;
}
```

The code is a bit unrealistic, because in a company with 100 employees, people are going to join or leave the company. The number of employees and their individual IDs are going to change. However, it is reasonable to assume that there could be a method called `getIDsFromDatabase`, which queries the corporate database and returns an array of `int` containing the ID of every employee who should get a paycheck. Then the preceding code would be modified as follows:

```
int[] ids = getIDsFromDatabase();
for (int id=0; id<ids.length; id++)
{
    float pay = getPayAmount(id);
    printCheck(id, pay);
    balance -= pay;
}
```


Multi-Dimensional Arrays

The arrays we have looked at so far have been *one-dimensional*. This means that each component is specified by a single unique index. Java also supports *multi-dimensional* arrays, in which each component is specified by a unique sequence of indices. The number of indices in the sequence is the array's *dimension*. In a two-dimensional array, for example, each component has two indices. Figures 6.1 and 6.2 portrayed some one-dimensional arrays as columns of boxes. We can portray a two-dimensional array as a lattice or matrix, where each component is identified by its row and column, as in Figure 6.3.

45	0
9	39
16	93

Figure 6.3: A two-dimensional array

If the array in Figure 6.3 were called `twoDimInts`, it would be declared as

```
int[][] twoDimInts;
```

The array is now declared but has not been created yet. When you create an n-dimensional array, you have to specify n sizes: one size for each dimension. To create a two-dimensional array of 3 rows and 2 columns, as in Figure 6.3, use the following code:

```
twoDimInts = new int[3][2];
```

Now the array has been created. At creation time, every component is initialized, as with one-dimensional arrays. To access an individual component, you use the array name followed by both indices, with each index in square brackets. For example, the following code initializes every component of `twoDimInts` to 39:

```
for (int i=0; i<3; i++)
  for (int j=0; j<2; j++)
    twoDimInts[i][j] = 39;
```

Suppose you have 50 weather stations, each of which takes a temperature reading every hour throughout one day. You might store the data in a two-dimensional float array called `temps`, where `temps[t][s]` is the temperature at time `t` recorded by station `s`.

The following code could be used to print the average temperature over all stations, hour by hour:

```
for (int hour=0; hour<24; hour++)
{
  float tempTotal = 0;
  for (int stn=0; stn<50; stn++)
    tempTotal += temps[hour][stn];
  float tempAvg = tempTotal / 50;
  System.out.println("Average temp at time " + hour +
    " = " + tempAvg);
}
```

On the other hand, you might want the average temperature over the entire day for each station. For that, you would use the following code:

```
for (int stn=0; stn<50; stn++)
{
  float tempTotal = 0;
  for (int hour=0; hour <24; hour++)
    tempTotal += temps[hour][stn];
  float tempAvg = tempTotal / 24;
  System.out.println("Average temp at station " + stn +
    " = " + tempAvg);
}
```

These examples show that processing a two-dimensional array generally requires a two-deep nested loop. In general, processing an N-dimensional array requires an N-deep nested loop.

The `BoolArrayLab` animated illustration uses a two-deep nested loop to let you set the values in a 200-by-200 boolean array. The array contents are illustrated by a grid of 200 by 200 pixels. A blue pixel represents a value of `true`; a black pixel represents `false`. (You probably can't see the individual pixels unless you use a magnifying glass.) To run the program, type `java arrays.BoolArrayLab`. The code looks like this:

```
boolean[][] bools = new boolean[200][200];
for (int y=0; y<200; y++)
  for (int x=0; x<200; x++)
    bools[x][y] = _____ ;
```

You supply the formula in the last line. The formula can be any valid boolean expression. Initially, the program comes up with the following formula:

```
bools[x][y] = x>y;
```

Figure 6.4 shows `BoolArrayLab`'s initial screen.



Figure 6.4: BoolArrayLab

The File , Gallery menu offers 7 sample formulas, and you are encouraged to try your own or modify the ones provided. [Figure 6.5](#) shows a parabola.

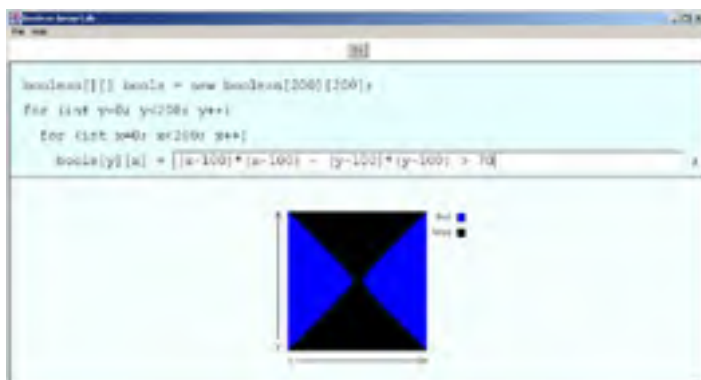


Figure 6.5: BoolArrayLab drawing a parabola

The lower portion of the display renders the contents of the array. Since 200 x 200 is fairly large, the display uses a rectangular grid of 200 x 200 pixels. An array component with a value of `true` is represented by a blue pixel, while a `false` value is represented by a black pixel.

Before you launch the program, can you guess what sort of image is produced by the initial formula `bools[x][y] = x>y;`? If you discover a formula that produces an interesting display, please email it to www.sybex.com. I will use it and mention your name in the next edition.

Arrays as Objects

Now you know enough about arrays to write some very useful code. At this point, we will stop discussing the syntax and use of array code. It's time to look at what you might think of as *array anatomy*. This material is extremely important, because nearly everything you'll learn about array anatomy also applies to the anatomy of full-blown objects. If you understand the material in the remainder of this chapter, you will have a good solid foundation for learning about object-oriented programming in Java.

You have already seen that declaring an array is different from declaring a primitive. When you declare a primitive, the variable is right there for you. But when you declare an array, you still have to create it. This is a clue that there is more going on with arrays than with primitives.

You can think of memory as being divided into two parts: *accessible* and *inaccessible*. (This is unofficial terminology, but it is very useful and will be used throughout this book.) Primitives exist in accessible memory; arrays exist in inaccessible memory. [Figure 6.6](#) shows memory divided into its two parts, populated with a few primitives and arrays.

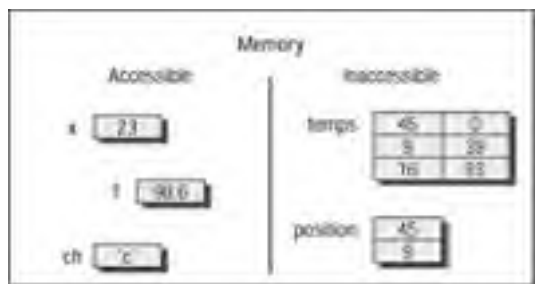


Figure 6.6: Accessible and inaccessible memory

Any variable on the left side of the figure – that is, in accessible memory – can have its values read and written. Accessible memory is for primitive variables, and for a kind of variable that you have already used without knowing it. This kind of variable is called a *reference*. References are used for all access to arrays. When you declare an array, what gets created is just a reference. (Remember, the array is not created until you say `new`.) The reference exists in accessible memory. No matter what kind of array you declare – no matter what type, size, or number of dimensions it has – the reference is 32 bits wide.

When you create an array by invoking the keyword `new`, space for the array is reserved (or *allocated*) in inaccessible memory. For example, the code `int[][] ages = new int[3][2];` would cause allocation of 24 bytes (3 times 2 ints, times 4 bytes per int), as shown in [Figure 6.7](#).

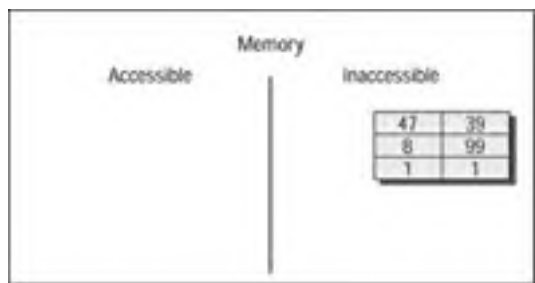


Figure 6.7: An array of bytes in inaccessible memory

Invoking `new` is a bit like invoking a method: The code returns a value. The value returned by `new` is almost – but not quite – the address in inaccessible memory of the freshly created array. If you are at all unclear about the distinction between memory address and memory contents, you might want to return to [Chapter 1](#) and play with the SimCom animated illustration.

Actually, it doesn't matter if the value returned by `new` is exactly the address of the array, or almost-but-not-quite the address, or only vaguely related to the address. The details of the relationship are a hidden part of the Java Virtual Machine, and they may even vary from one implementation of the JVM to another. For this reason, the value is called a *reference* to the array. *Reference* implies that the value uniquely identifies the array in a way that is hidden from us.

Now we can look at what really happens when the following code is executed:

```
int[][] ages; // Allocation
ages = new int[3][2]; // Construction & ref assignment
```

The allocation line creates a reference named `ages` in accessible memory. Then the creation line causes space for the array to be allocated in inaccessible memory. The invocation of `new` returns a reference to the array; this reference is the right-hand side of the `=` assignment. The reference is then stored in the variable whose name appears on the left side of the `=` assignment, namely `ages`. This situation is illustrated in [Figure 6.8](#).

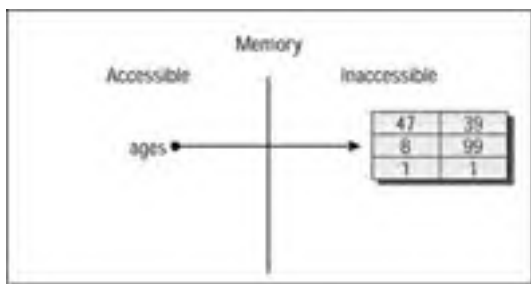


Figure 6.8: Reference and array

Notice that the comment on the second line of code above says *Construction* rather than *Creation*. *Construction* is the technical name for creating something in inaccessible memory by invoking `new`.

Why does this matter? So far, the array code we have presented has made sense without burdening you with all this reference stuff. But there are certain very useful operations you can do with arrays that only make sense if you understand references. These are operations that you have already seen in the context of primitives: assignment, and argument passing.

Suppose `ages` and `otherAges` are declared to be arrays of the same type. What does it mean to say the following?

```
ages = otherAges;
```

Contrary to reasonable expectation, this code emphatically does *not* create a new array whose components have the same values as the original array. Remember that `ages` and `otherAges` are really references. So `ages = otherAges;` just copies the 32-bit pattern from one reference to the other. The result is a second reference that (in some sense) points to the same thing the first reference pointed to: the array. You now have two references to the same array, as shown in Figure 6.9.

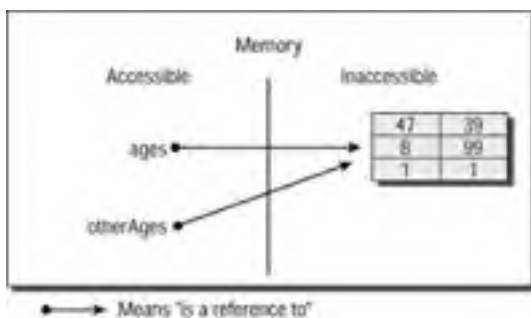


Figure 6.9: Two references, one array

The `CreateArrayLab` animated illustration dynamically illustrates the following code:

```
double d = 1.23;
double e = d;
d = 3003;
double[] doubleArray = new double[4];
double[] theCopy = doubleArray;
doubleArray[1] = 98.6;
e = theCopy[1];
```

Start the program by typing `java arrays.CreateArrayLab`. You will see the display shown in Figure 6.10.

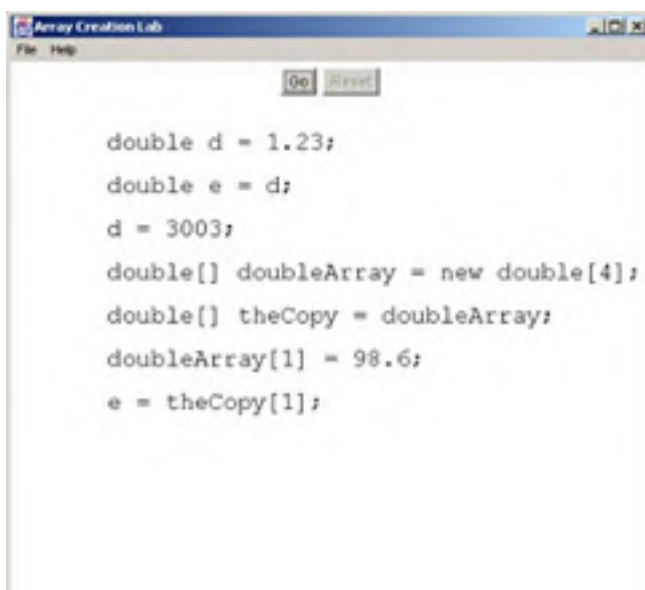




Figure 6.10: CreateArrayLab

Click on the Go button to see the animation. When the animation finishes and you want to watch it again, you can click on Reset to return the display to its original state.

References usually point to arrays (or to objects, as we will see later). However, there is a special value that can be assigned to any reference. The value is `null`, and it indicates that the reference does not point to anything. For example, you might make the following declaration and assignment:

```
double[] doubles;  
doubles = null;
```

The `null` reference value is only slightly useful at the moment, but you will see a use for it in the section on garbage collection later in this chapter. You will also make extensive use of `null` in later chapters, in the context of objects.

Passing References to Methods

Now you understand what really happens in code like the following:

```
float[] floatArray = new float[4];  
float[] theCopy = floatArray;
```

Are you likely to encounter this situation? Are you likely to make a copy of a reference, when you already have a perfectly good one, given the confusion that a copy might create? Actually, yes. Within a single method, there isn't really any good reason for copying a reference. However, you might want to pass an array as a method argument.

Remember that when you pass a primitive as an argument to a method, the method actually gets a copy of the primitive. Thus, the method can modify the copy, and the caller will never be aware of the modification because the caller has no access to the modified copy.

With arrays and methods, the situation is a bit different. You don't actually pass an array into a method. You pass a reference to the array. The method receives a copy of the reference. The caller's original reference and the method's copy are identical 32-bit patterns, so they both (in some sense) point to the same object in inaccessible memory: the array. So when you pass an array reference as a method argument, the method can use the reference to modify the array, and the modifications *will* be visible to the caller.

The PassArrayLab animated illustration animates the following code:

```
int[] intarray = new int[3];  
setInts(theArray);  
.  
.  
static void setInts(int[] ints)  
{  
    for (int n=0; n<ints.length; n++)  
        ints[n] = 22;  
    return;  
}
```

The `return` statement isn't really required, since the method's type is `void` and it would return anyway after executing its last line. The `return` is just there to make the animation more clear. When a method returns, all variables that were declared within the scope of the method cease to exist. This includes the method's arguments (`ints` in this example). The space in accessible memory that was allocated for the variables is reclaimed by the system. Conceivably, the next variable to be declared could occupy the same bytes that used to constitute the `ints` argument. But all is not lost. Although the `ints` argument ceases to exist, the array it references continues to exist.

Invoke PassArrayLab by typing `java arrays.PassArrayLab`. The Go and Reset buttons start the animation and reset the display. Run the animation a few times, until you are confident that you understand that what gets passed to the method is a reference and not an array. That way, changes made to the array are permanent and visible to the caller.

Garbage Collection

Garbage collection is a mundane term for a very important feature.

You have seen that arrays are created by invoking the keyword `new`. Surely there must be a way to recycle an array's memory after the array is no longer needed. Something like this happens to the arguments and local data in a method, when the method returns. But in that case, the recycled data consists of primitives and references. In other words, it's data in accessible memory that was reserved by declaring arguments or variables. In the case of arrays, we are concerned with data in inaccessible memory that was reserved by invoking `new`.

Java's precursor languages, and in particular C and C++, required the programmer to explicitly free up memory that was no longer needed. This was the only way that the memory could become available for reuse. This led to problems. One such problem is called a *memory leak* bug. If a bug causes a method to neglect to free up a few hundred bytes, that's not much of a problem. The

program is probably running on a system with at least several million bytes of available memory, so if a few hundred become unavailable, there is still plenty left. But if the method is called in a loop that executes 1,000 times, a few hundred thousand bytes become unavailable. That might have an impact. If the loop executes 1,000 times every hour, eventually there will be no more memory available for allocating new arrays, and the program will crash. The problem is called a memory leak because the pool of available memory gradually diminishes.

Java makes memory leaks highly unlikely, because in Java the programmer never decides when to recycle unneeded arrays and objects. The JVM decides when memory is no longer needed, and such memory is automatically recycled. The JVM uses the following logic to decide when to recycle memory:

When an array (and, as we'll see later, an object) is created, inaccessible memory is created and a reference is returned. The JVM keeps track of how many references are pointing to an array or other object. Your program might make copies of a reference, and might pass the reference as a method argument. As long as there is at least one reference to something, there is a chance that you might want to use that particular something, so its memory will not be recycled. However, when the last reference to an object ceases to exist, suddenly there is no way to read or write the object, or to use it in any way. You can't talk about something if you have no name for it. Since the unreferenced object can no longer play any role in your program, its memory will be automatically recycled. Such automatic recycling of unneeded memory is called *garbage collection*.

Consider the following method:

```
1. void useAnArray(int size)
2. {
3.     int[] theArray = new int[size];
4.     int[] aRefCopy = theArray;
5.     int[] anotherCopy = theArray;
6. }
```

Line 3 constructs an array of ints and stores the returned reference in `theArray`. Line 4 copies the reference, so after line 4 there are 2 references to the array. After line 5, there are 3 references to the array, but only briefly. Immediately after line 5, the method returns, so all its variables are recycled. Suddenly, instead of 3 references to the array, there are none at all. Now the program no longer has any way to access the array, which will be recycled shortly.

If you have an array that you no longer need, there is no explicit way to recycle its memory. However, you can usually set all references to the array to `null`, which will cause garbage collection.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. The following two declarations are equivalent as far as the compiler is concerned, but one is considered more readable than the other. Which is more readable, and why?
 1. `double dubs[];`
 2. `double[] dubs;`
2. Write a line of code that declares an array of 5 ints and initializes the array to contain the first 5 prime numbers. The code should be a single statement.
3. Write a method whose single argument is an array of double. The method should return the average (mean) of the array's components. Write an application that tests the method by passing it an array containing any values you like.
4. Write a program that uses the array-averaging method of Question 3. The program should compute and print out the average of an array (you can choose the component values). Then the program should add 100 to each component, and again compute and print out the average.
5. Write a program that contains a method that creates and returns an array of int containing the first n square numbers, where n is the method's argument. Test your method by calling it with $n=10$. Your program should print out the index and value of each component, in *descending* order.
6. Write a method that creates a multiplication table. The method should return a two-dimensional array of N by N ints, where N is specified by the method's argument. In the array, the component at `[row][col]` should have a value of `row*col`.

Chapter 7: Introduction to Objects

The [previous chapter](#) presented arrays, which are Java's simplest kinds of objects. Arrays are much less sophisticated than other kinds of objects. Usually when people say "object," they mean it in a casual sense that excludes arrays. A program whose only objects are arrays can hardly be called object-oriented. However, in learning about arrays, you have learned a number of concepts that are vital to your mastery of full-fledged objects. You are now ready to enter the world of object-oriented programming, perhaps never to return.

The animated illustrations for this chapter provide visual reinforcement for the concepts that will be presented here. Please be sure to run them and take the time to play with them when the text invites you to do so.

Arrays Versus Objects

Before we begin, let's agree on some terminology. In the most formal sense, an array is a kind of object. However, we are about to compare arrays and other objects, and we need to avoid cumbersome language. It would be useful to say, for example, "objects have data and methods, rather than, "objects that aren't arrays have data and methods." So for the remainder of this book, unless it will cause confusion, "object" will mean "object but not array."

You already know a lot about objects from your study of arrays. Here are some similarities between arrays and objects:

- Objects contain clusters of data.
- Objects are created by invoking the keyword `new`.
- Objects inhabit inaccessible memory.
- Objects are manipulated indirectly, via references.
- Object references can be passed as method arguments; objects cannot.
- Objects are not explicitly destroyed; they are garbage-collected when they have no more references.

Another recognizable feature of objects is the use of the period as a symbol to denote "property of" when it follows the name of an array or object. With arrays, the syntax `arrayReference.length` gave you the number of components in the array. With objects, the syntax `objectReference.something` gives you access to the extensive power and features of an object. You will see how this works in great detail later in this chapter.

Objects have many features that go far beyond what arrays can do. Here are some unique features of objects:

- They can contain data of different types.
- They can contain methods as well as data.
- They are related to classes.

Classes are among the most important concepts in object-oriented programming. They are actually quite simple to understand, as you will see in the [next section](#).

Classes

Plato would have approved of the concept of classes.

The easiest way to learn about classes is to step outside the domain of object-oriented programming for a moment and look at the real world. (Plato might not have approved of calling it "real.") We experience things in the world, and we create categories in our minds so that we can think about those things collectively. [Figure 7.1](#) illustrates this mental process.

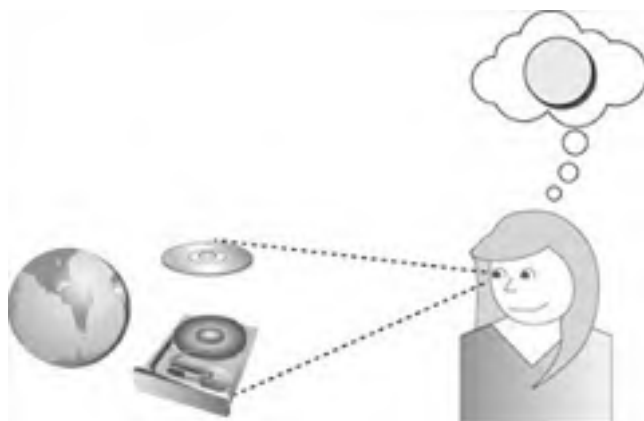


Figure 7.1: Class as mental category

As another example, "dog" is a category. In daily life, when dealing with the external world, we don't really experience the category "dog." We experience individual dogs, such as Harley or Sumo or Rover. So "dog" is a class or category, and the individual dogs Harley and Sumo and Rover are individual instances of that class.

Now back to object-oriented programming. In Java, a *class* is a piece of code that describes a category of thing that you want to represent in software. In fact, a Java program is just a bunch of interacting class definitions. You might have suspected as much, since every complete application listing we have seen so far has contained the mysterious keyword `class`. On the other hand, you might *not* have suspected as much. So far we have put a lot of effort into concealing the object-oriented nature of class code, because it was not yet time to talk about objects and classes. Well, now the time has come.

Suppose you want to write a program to model the behavior of Harley, Sumo, and Rover. First you spend some time thinking about what these three have in common. Eventually you realize that they are all dogs, so you decide to create a class called Dog. (In Java, a class name can be any valid identifier, but by convention we capitalize the first letter.) You create the class by writing a source file that looks like this:

```
public class Dog
{
    . . .
}
```

This is called a class definition. The code that goes between the curly brackets is the *body* of the class definition. Bodies can be as short as a few lines of code, for very simple classes. There is no upper limit on the size of the body, but typical large class bodies can be hundreds or even thousands of lines long. Of course, you don't yet know how to write a class body, but that is what the rest of this book is all about.

A class definition should appear in a file whose name matches the class name. So the Dog class should appear in a file called `Dog.java`. Compiling this file will result in a file called `Dog.class`. Now that you know what a class is, the `.class` filename extension makes sense. This is not an absolute rule, but explaining when you do and don't have to apply it would require presenting a number of concepts that are out of place here. If you are curious, please wait until [Chapter 9, "Packages and Access."](#)

So a class is something that you define when you write your source code. What about objects? An *object* is an individual instance of a class. Objects are created when your program is executed. More specifically, an object, like an array, is created by an invocation of the keyword `new`. The syntax for object creation is a bit different from the syntax for array creation, as you will see in the [next section](#).

Objects and Their Data

You have already learned that objects, like arrays, contain clusters of data. You have also learned that the data in an object, unlike the data in an array, can be of differing types. The number, types, and names of an object's data elements are all defined in the object's class definition.

Let's look at a very simple example. Here is a very simple class definition:

```
public class Person
{
    int    age;
    short weight;
}
```

This is our first example of a class that does not contain a method called `main`. In fact, this class definition has no methods at all. The class just defines a bundle of data. The bundle contains an `int` called `age` and a `short` called `weight`.

To create an individual instance of this class, you would use the following code:

```
Person keara;
keara = new Person();
```

The first line is a declaration. Like all other declarations, it tells the compiler that you will be using a variable called `keara` and it will be of a certain type. At first glance, it appears that the type of `keara` will be `Person`; this is almost true, but not quite. Actually, the declaration says `keara` will be a *reference* to an object, and that object will be an instance of the `Person` class. If the distinction seems subtle, it is also very important.

The situation is similar to what we saw in the [previous chapter](#), in the context of arrays. The declaration `int[] temperatures;` says that `temperatures` will be a reference (in accessible memory) to an array (in inaccessible memory). Similarly, the declaration `Person keara;` says that `keara` will be a reference (in accessible memory) to an array (in inaccessible memory). In both cases, the declaration does not cause construction of the array or object. Nothing gets constructed until `new` is executed.

When the second line (`keara = new Person();`) executes, an object is constructed, using the class definition as a kind of stencil or cookie cutter. The JVM knows which class definition to use (remember, a Java application can consist of many class files) because the class name appears after `new` and before the empty parentheses. As with arrays, the invocation of `new` constructs an object in inaccessible memory and returns a reference to that object. The reference is stored in the variable `keara`, so `keara` now refers to the newly created object. At this point, the situation is as shown in [Figure 7.2](#).

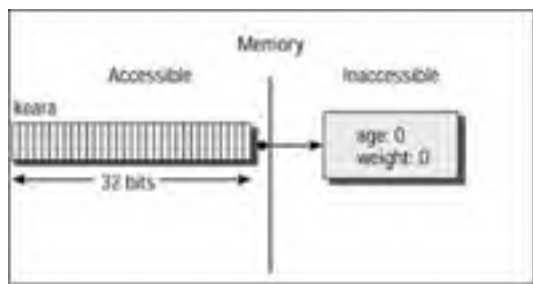


Figure 7.2: Reference and object

The figure shows that the object contains its own set of data variables, with names and types as specified in the class definition source file. When an object is constructed, its data variables (called *fields*) are initialized in the same way array components are initialized. Numeric fields are initialized to 0, char fields are initialized to the null character, and boolean fields are initialized to false.

Now you can use the reference `keara` to manipulate the object's fields. The following code writes and reads the fields of an object, using a new notation:

```
keara.age = 8;
short f = keara.weight;
```

In both lines, you use the following syntax to refer to an object's field:

```
Object_reference.field_name
```

The period is pronounced "dot." So if you were reading the first code line out loud, you would say, "Keara dot age equals eight."

The DataLab animated illustration shows the construction of an object and the use of a reference to access fields of that object. Please take a moment now to run the animation by typing `java objects.DataLab`. [Figure 7.3](#) shows DataLab's initial display.



```
public class Person
{
    int age;
    float weight;
}

// ...

Person pears = new Person();
pears.age = 32;
pears.weight = 150;
// ...
```

Figure 7.3: DataLab

Press the "Run" button to view the animation.

Team LIB

← PREVIOUS

NEXT →

Multiple Objects

You have learned that a class is a kind of stencil or cookie cutter for creating objects. Cookie cutters are especially useful if you're going to make lots of cookies. Similarly, classes are really useful because you can use a class to make multiple instances of that class. Different cookies made from the same cutter have the same outline, but they can be frosted or decorated differently. Similarly, different instances of the same class can have different data values.

Figure 7.4 shows three instances of the Person class. Each instance is referred to by a different reference.

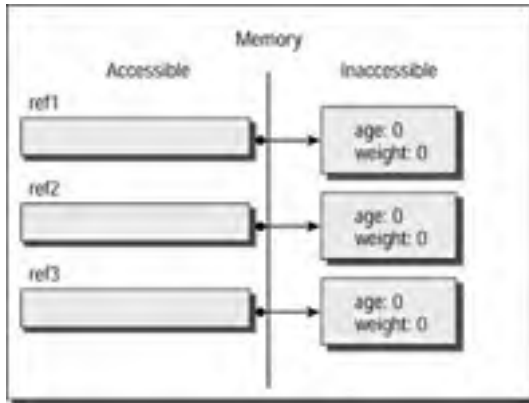


Figure 7.4: Multiple objects

The data values inside an object can be manipulated in all the same ways you can manipulate ordinary variables. For example, if `curly` and `larry` are references to Person objects, you might write the following:

```
curly.age = larry.age + 12;
```

The SeveralObjectsLab animated illustration lets you play with multiple instances of the Person class. Start the program by typing `java objects.SeveralObjectsLab`. You will see the display shown in Figure 7.5.



Figure 7.5: SeveralObjectsLab

SeveralObjectsLab initially displays the following code:

```
Person reference1 = new Person();
Person reference2 = new Person();
Person reference3 = new Person();
reference1.age = 30;
reference1.age = 30;
reference1.age = 30;
reference1.age = 30;
reference1.age = 30;
```

The reference variable names aren't very imaginative. Also, it isn't very useful to have five lines that all set the same field in the same object to the same value.

That's where you come in. Type better names into the text fields in the declaration lines. Use the choices to reference different fields in different objects. Use the text fields in the assignment lines to assign any value you like to the fields you have chosen. The assignment values can be literals or expressions, and the expressions can include fields in any of the objects. Figure 7.6 shows SeveralObjectsLab after reconfiguring.



Figure 7.6: SeveralObjectsLab reconfigured

Figure 7.7 shows the result of executing Figure 7.7.



Figure 7.7: SeveralObjectsLab reconfigured and executed

Try configuring SeveralObjectsLab to execute the following code:

```
Person simon = new Person();
Person emily = new Person();
Person bethan = new Person();
simon.age = 30;
emily.age = simon.age - 20;
simon.weight = 150;
bethan.weight = simon.weight / 3;
bethan.age = bethan.weight / 5;
```

The program demonstrates construction of the objects and references, followed by assignment to the various fields. Try configuring different values. Hopefully, the results will not be surprising.

The point of SeveralObjectsLab is to get you to think of objects as bundles of data. Objects are similar to other instances of the same class, to the extent that all such objects contain similar clusters of data. That is, each cluster has the same number of variables, and those variables have the same types and names, as defined in the class definition file. However, each object is distinct and has its own version of each variable defined by the class.

Objects and Their Methods

In addition to containing data, objects can also contain methods. You might have suspected as much, because all the application classes presented in this book have contained at least one method (`public static void main(String[] args)`), and sometimes more than one. All those methods have had the keyword `static` in their declarations. Later in this chapter you will see that `static` means, in a sense, "not object-oriented." The methods had to be static because we had not yet introduced objects. Now it is time to present genuinely object-oriented methods.

Let's add a method to the `Person` class:

```
public class Person
{
    int    age;
    short  weight;

    int ageInNYears(int n)
    {
        return age + n;
    }
}
```

The method computes how old the person will be in `n` years. It has a lot in common with the methods you have already seen. It has a declaration that specifies the method's return type, name, and argument list. It has a body enclosed in curly brackets, and it returns a value.

There are two major differences between this method and the ones you have looked at in previous chapters:

- There is no `static` in the declaration.
- The method refers to a field (`age`) of the class where the method is defined.

To call a method of an object, you again use the "reference-dot" notation. The following code shows how this is done:

```
1. Person ed = new Person();
2. ed.age = 62;
3. ed.weight = 220;
4. int n = ed.ageInNYears(3);
5. System.out.println("Ed will be " + n +
    " in 3 years.");
```

Note in line 4 that the method call looks like the method calls you are used to, except that it is preceded by an object reference and a dot. This syntax says, "Call the `ageInNYears` method of the object referenced by `ed`."

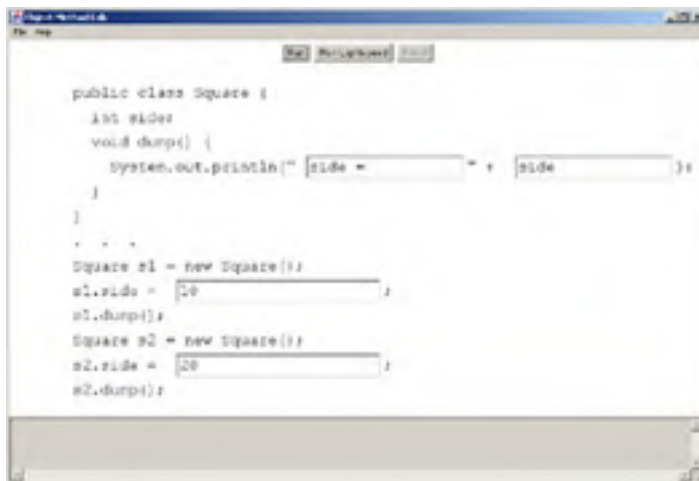
This brings up an important point about object-oriented programming. Until this chapter, all the methods presented in this book have contained in their declarations the mysterious keyword `static`. We will explain `static` in detail in the [next section](#). For now, let's just say that a static method is one that is not object-oriented and does not belong to an individual object. It has been useful to present only static methods for six chapters, because the non-static approach allowed us to introduce a great many foundational concepts without the added complication of presenting objects. But in realistic Java programming, very few methods are static. Most methods are non-static, which means they are associated with objects. Thus, most method calls involve not just invoking a method, but invoking a method *on an object*.

Let's look again at the `Person` class, with line numbers:

```
1. public class Person
2. {
3.     int    age;
4.     short  weight;
5.
6.     int ageInNYears(int n)
7.     {
8.         return age + n;
9.     }
10. }
```

The `ageInNYears` method makes use of the `age` variable. But *which* `age` variable? Every instance of the `Person` class has its own version of `age` (and of `weight`). You may have already guessed correctly: The version of `age` that gets used is the one belonging to the object on which the method call was made. So in the line `int n = ed.ageInNYears(3);`, the version of `age` that gets used is the one belonging to the object referenced by `ed`.

The `ObjectMethodLab` animated illustration demonstrates a class called `Square`. Start the program by typing `java objects.SeveralObjectsLab`. You will see the display shown in [Figure 7.8](#).



```
public class Square {
    int side;
    void dump() {
        System.out.println("side = " + side);
    }
}

Square s1 = new Square();
s1.side = 10;
s1.dump();

Square s2 = new Square();
s2.side = 20;
s2.dump();
```

Figure 7.8: ObjectMethodLab

The class contains one variable, an `int` called `side`. The class also has one method, called `dump`, which outputs a message followed by a value. The output appears in the text area at the bottom of the window. Initially, the method dumps out `side =`, followed by `side` itself. Of course, this is the version of `side` that is owned by the object on which the method was called. The code creates two objects and gives them distinct values for `side`. These values are 10 and 20, but you can change them by typing different numbers into the text fields.

Try configuring the code so that the method dumps the perimeter of the square. Configure again so that the method dumps the area of the square. Observe how, when you call `dump` on an object, the method uses that object's version of `side`.

The [next section](#) will look deeper into how objects contain interacting data and methods.

The Truth About *static*

The time has come to show you how classes, objects, static code, and non-static code all work together to create a complete object-oriented Java application program. To understand what's going on in an application, you need to be clear on the difference between static and non-static parts of a class. You have already learned that methods can be either static or not, and that most methods in most programs are non-static. Data, as well as methods, can be either static or not. Let's look at this distinction.

Static Data

The [previous section](#) explained that when a class defines data members, each instance of the class gets its own version of each data member. This is true for ordinary non-static data. If you add `static` to the declaration of a variable in a class, you defeat this one-version-per-instance mechanism. Instead of getting one version of the variable for each instance, you just get one version of the variable, period.

Here's a version of the `Person` class that has a static variable:

```
1. public class Person
2. {
3.     static int    rev = 3;
4.         int    age;
5.         short  weight;
6.
7.     int ageInNYears(int n)
8.     {
9.         return age + n;
10.    }
11.
12.    void dump()
13.    {
14.        System.out.println("rev " + rev +
15.                            " age = " + age);
16.    }
17. }
```

Static variables have limited uses. One possible use is to keep track of the current revision of the class source. Here you set the `rev` to 3. (As with any other variable declaration, you can initialize a static class variable in the same line where you declare it.) The `rev` is 3 because version 1 from earlier in this chapter just had data, `rev 2` from the [previous section](#) had data and a method, and that brings us to `rev 3`.

When the `dump` method executes, the version of `age` that gets printed out is of course the version belonging to whatever `Person` object is executing the method. The version of `rev` that gets printed out is... well, there is only one version, because the variable is static. Consider the following example:

```
Person thelma = new Person();
thelma.age = 28;
Person louise = new Person();
louise.age = 38;
thelma.dump();
louise.dump();
```

The output of this code is

```
rev 3 age = 28
rev 3 age = 38
```

The (static) `rev` does not change but the (non-static) `age` does.

Now consider the following code, which uses static data to get into trouble:

```
Person thelma = new Person();
thelma.age = 28;
Person louise = new Person();
louise.age = 38;
thelma.rev = 999; // Change thelma's rev
louise.dump();
```

Now the output is

```
rev 999 age = 38
```

If you didn't know that `rev` was static, you would be surprised by the output. The code seems to change the `rev` of `thelma`, not of `louise`. Of course, since `rev` is static, it doesn't belong to an individual object, so it isn't really meaningful to talk about the `rev` "of `thelma`" or "of `louise`." There is just the `rev`.

Java offers you a way to refer to static variables without risking the confusion of the previous example. Instead of saying `thelma.rev` or `louise.rev`, you can say `Person.rev`. In other words, instead of the reference-dot-staticVariableName syntax, you can use `classname-dot-staticVariableName`. This makes static variable usage more conspicuous, because typical class names begin with capital letters, while reference names begin with lowercase letters. (This is not a requirement of the language; it is a style convention. There is no benefit to violating this convention.)

Using the new syntax, the previous example can be rewritten as


```
Person thelma = new Person();
thelma.age = 28;
Person louise = new Person();
louise.age = 38;
Person.rev = 999;
louise.dump();
```

The change makes it clear that the thing that's getting set to 999 is a static variable, so the output should now be no surprise to anyone. The `classname-dot-staticVariableName` notation reinforces the fact that the variable does not belong to any instance. It is convenient to think of statics as belonging to the class as a whole, rather than to an individual instance. By contrast, non-static variables are sometimes called *instance variables*.

We can now move from static data to the more subtle concept of static methods.

Static Methods

You have just seen that a static variable is not associated with an individual object. Similarly, a static method can be thought of as acting in a way that is not associated with an object.

In an instance method (that is, a non-static method), access to an instance variable meant access to the version of the variable owned by the currently executing object. Thus, in the `Person` class, the `ageInNYears` method used the `age` variable. Calling the method on `thelma` meant that the method would use `thelma`'s age. Calling the method on `louise` meant that the method would use `louise`'s age. In all cases, the method used the current object's version of the variable.

In a static method, there is no current object, so it would be meaningless for a static method to use a non-static variable. (Which object's version of that variable should be used? There's no good answer.) A static method is not allowed to read or write the non-static data of its class. Also, a static method may not call the non-static methods of its class.

Sort of.

To really understand what static code can and cannot do, you need to know about a useful Java feature called the *this-reference* notation.

Earlier in this chapter, you learned about the `reference-dot-variableName` and `reference-dot-methodName` notations. These constitute the grammar that lets Java be object-oriented. In object-oriented programming, you specify not only what data or method you want to access, but the object that owns the data or method. But within the instance methods we have seen, instance variables have been accessed without the reference-dot notation. The `ageInNYears` method returned `age + n`, and there is no reference-dot notation there.

In an instance method, any use of a variable without a reference-dot prefix is something like an abbreviation. For example, the method

```
int ageInNYears(int n)
{
    return age + n;
}
```

can be thought of as an abbreviation of

```
int ageInNYears(int n)
{
    return this.age + n;
}
```

The keyword `this`, also known as the *this-reference*, is a reference to the current object. So if you called `thelma.ageInNYears(20)`, within the method, `this` would reference the same object `thelma` referenced.

A static method has no *this-reference*, so it cannot use the abbreviated notation enjoyed by non-static methods. A static method can indeed access non-static data and methods of its own class, or of any other class, but the method must explicitly provide a reference to the intended object. So the following would be perfectly legal:

```
static void printLouisesAge(Person louise)
{
    System.out.println("Louise is " +
        louise.age);
}
```

We can summarize all this static/non-static information as follows:

- A non-static method may use non-static data and methods of its class without using the reference-dot notation. The current object is implied.
- A static method must use the reference-dot notation. There is no current object.
- A static method has no *this-pointer*.

Now at last, we can tie everything together and explain the role of the static `main` method.

The *main* Method

You have seen that static features of a class are a way of getting around the object-oriented requirement that data must live inside objects and methods must be called on objects. Ideally, an object-oriented program would be a federation of objects of many different classes that make method calls on one another, creating new objects as needed and allowing old ones to be garbage-collected when no longer needed. This image is fine once the application is up and running, but how does the process get started? If objects are constructed by other objects invoking `new`, how does the first object get created?

In Java, everything starts with the `main` method. Through the end of the [last chapter](#), you patiently tolerated the presence of `static` in the `main` method's declaration. Now you know what it means: `main` is not called on any individual object. It is static, so

it is just called. Within `main`, objects can be created and non-static calls can be made, so the object interactions quickly become highly object-oriented.

Every application is invoked by typing

```
java ApplicationClassName
```

(The animated illustration programs require a prefix and a dot before the class name. We will explain that notation in [Chapter 9](#).) When you start up an application, a program called `java` is executed. This is the Java Virtual Machine. The JVM does *not* create an instance of the application class. It could have been designed to do so, but the creators of Java decided to let us programmers decide when and how to create objects.

After the JVM initializes itself, it makes use of one of its parts, called the *class loader*. The class loader is code that finds, reads, and interprets `.class` files. After the class loader processes a `.class` file, the JVM is changed in two ways:

- The class defined in the file can be used by the JVM.
- Any static data declared in the class is allocated and initialized.

Initially, the JVM uses the class loader to load the class specified in the command line. Later on, during the course of execution, any class used in the code that has not been loaded already is loaded as needed. Since the class loader allocates and initializes static data before any instances of the class are constructed, you can access a class's static data, and even call its static methods, even if no instances of the class exist.

That's good news, because the next thing the JVM does is call a static method of the class it just loaded. Of course, this is the `main` method. Presumably, `main` constructs objects that construct objects that construct objects, and the program enters its object-oriented phase. Static data and methods can still be used, but typically most accesses are non-static.

The `ObjectLifeCycleLab` animated illustration demonstrates how static code starts the chain of object-oriented interactions. Start the program by typing `java objects.ObjectLifeCycleLab`. At first the display only shows a star, representing the static `main` method of an application. This initial state is shown in [Figure 7.9](#).

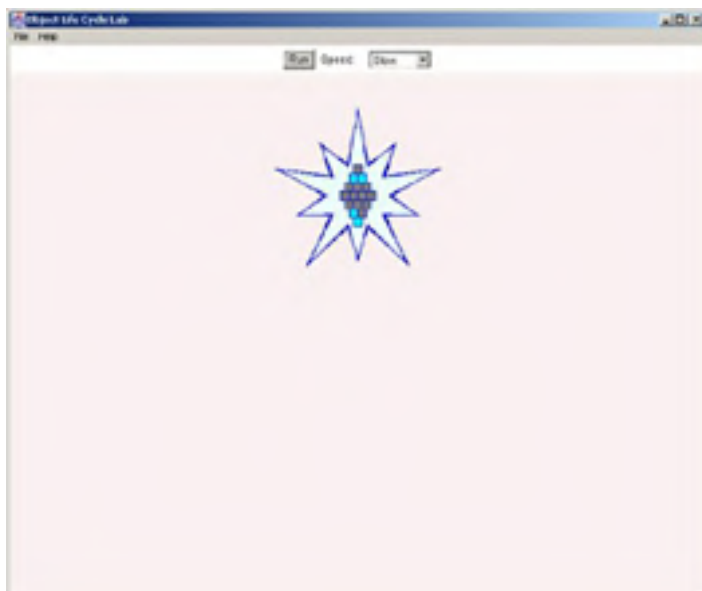


Figure 7.9: `ObjectLifeCycleLab`

When you click the `Run` button, the static code constructs an object that is an instance of one of three classes: `Triangle`, `Rectangle`, and `Oval`. Each object contains its own data and methods. The static code makes a method call on the object, represented by an expanding arrow. The colored dots near the arrowhead represent method arguments.

Now the code in the first object's method constructs a second object, on which a method call is made. The call is returned (that colored dot near the shrinking arrowhead represents a return value), and the life cycle goes on and on and on. Sometimes objects vanish; this represents garbage collection. [Figure 7.10](#) shows `ObjectLifeCycleLab` after running for several minutes.





Figure 7.10: ObjectLifeCycleLab after running a while

Of course, ObjectLifeCycleLab is just a symbolic cartoon, but it illustrates several very important concepts. Watch the program until you observe the following behaviors:

- Everything starts with static code, all alone.
- At any moment, the static code or a single object is *current* (recognizable by a highlighted background and flashing data).
- An object only gets garbage-collected if it is not in use.

Reference Data

Variables in a class don't have to be primitive. Classes may define variables that are references to objects or to arrays. Moreover, arrays may contain references rather than primitives. When an object with reference data is constructed, the references are all initialized to the `null` value, indicating that they do not yet point to any objects.

For example, suppose you are writing a Java program to control a weather station that has electronic access to a number of remote thermometers. You might model this situation with two classes, `WeatherStation` and `Thermometer`.

Writing the code that would enable the `Thermometer` class to read input from a physical device is far beyond the scope of this book. Let's just assume that somehow the class has a method called `connect`, which takes care of the connection, and another method called `readTemp`, which returns a float.

The `WeatherStation` class would include the following data declarations:

```
Thermometer[]    therms;
```

As with any other array, the declaration does not create the array. You would create the array in a method, with a line like the following (assuming there are 20 thermometers):

```
therms = new Thermometer[20];
```

Now the array exists, and all its components are `null`. You now have to construct and connect each `Thermometer` object, as follows:

```
for (int i=0; i<therms.length; i++)
{
    therms[i] = new Thermometer();
    therms[i].connect();
}
```

Now you can write a method to compute the average temperature:

```
float getAverageTemp()
{
    float totalTemp = 0;
    for (int i=0; i<therms.length; i++)
        totalTemp += therms[i].readTemp();
    return totalTemp / therms.length;
}
```

Let's refine the `connect` method to illustrate the use of `null`. Sometimes hardware fails. Let's assume that `connect` can detect a failure of the thermometer belonging to the executing object. This failure will be indicated by the method's return value: `true` will mean connection was successful, and `false` will mean there was some kind of failure. You can rewrite the array initialization code like this:

```
for (int i=0; i<therms.length; i++)
{
    therms[i] = new Thermometer();
    if (therms[i].connect() == false)
        therms[i] = null;
}
```

Now you need to refine the `getAverageTemp` method so that it ignores all broken thermometers:

```
float getAverageTemp()
{
    float totalTemp = 0;
    int nWorkingThermometers;
    for (int i=0; i<therms.length; i++)
    {
        if (therms[i] != null)
        {
            totalTemp += therms[i].readTemp();
            nWorkingThermometers++;
        }
    }
}
```

```
    }  
  }  
  return totalTemp / nWorkingThermometers;  
}
```

You use the variable `nWorkingThermometers` to count `Thermometer` objects that actually contributed to the average.

A reference whose value is `null` may not be used for accessing an object. (After all, `null` means that there is no object pointed to by this reference.) For example, the following is illegal:

```
Thermometer thermo = null;  
Float temp = thermo.readTemp();
```

When the second line is executed, the program is terminated abruptly with an error message about a null pointer exception. You have already seen exceptions, but we will not discuss them in detail until [Chapter 11, "Exceptions."](#) The name "null pointer" is a throwback. Java's predecessor languages, C and C++, use pointers, which are like references but less secure. It isn't clear why the exception was named "null pointer exception" rather than "null reference exception."

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Name four traits that arrays and objects have in common.
2. Name two differences between arrays and objects.
3. Objects are not passed as method arguments, but references to objects can be passed. When a reference is passed into a method, any changes made to the referenced object by the method should be visible to the method's caller. Write an application to demonstrate this.

Your application will have two classes: `Cat` and `Ager`. The `Cat` class should have a single variable: an `int` called `age`. The `Ager` class should have a method whose signature is `makeOlder(Cat kitty, int nYears)`. This method should add `nYears` to the age of the `Cat` object referenced by `kitty`. Your `main` method should go in the `Ager` class. It should create one instance of each class, set the cat's age, and then use the `Ager`'s method to change the age. Your `main` should then print out the cat's new age, and verify that it really changed.

4. What happens if you move the `main` method of the previous question from the `Ager` class to the `Cat` class?
5. Write an application that causes a "null pointer exception" failure.
6. What does the following application print out?

```
public class Question
{
    static long x;

    public static void main(String[] args)
    {
        Question q1 = new Question();
        Question q2 = new Question();
        q1.x = 10;
        q2.x = q1.x + 20;
        System.out.println("q1.x = " + q1.x);
    }
}
```

Chapter 8: Inheritance

In the [last chapter](#), you learned that objects are instances of classes, containing data and methods defined by the class. This chapter will present two object-oriented concepts that will greatly enhance what you can do with objects. At first glance, inheritance and constructors do not seem to have much to do with each other. However, by the end of this chapter, you will see that a class's constructors are intimately related to that class's inheritance hierarchy.

Superclasses and Subclasses

You already know that a class has data and methods. You provide a class with these features by writing the code that defines the class. Now it's time to learn another way that a class can get data and methods: *inheritance*.

Inheritance is how a class can get data and methods that are defined in a different class. For this mechanism to work, the two classes must have a special relationship with each other: one must be a *superclass* of the other, which must be a *subclass* of the first. In this section, you'll learn what this relationship means.

Let's start with an example. Suppose you are writing a Java program to support the personnel department of the company. You decide that you should create two classes to represent the employees: `Worker` and `Manager`. These classes have some similarities and some differences. Here are some of the similarities:

- Workers and managers both have employee identification numbers, so both classes have an `int` called `id`.
- Workers and managers both need to get paid, so both classes have a `float` called `salary` and a method called `printCheck`. (The details of creating a method that prints checks are beyond the scope of this book, but it seems only fair that everybody should get a check.)

Now here are some of the differences:

- Managers have workers who report to them, so the `Manager` class has an array of `Worker` objects called `workers`. Workers don't need this, because nobody reports to them.
- Workers might or might not be eligible for overtime pay, so the `Worker` class has a `boolean` called `getsOvertime`. Managers are never eligible for overtime, so the `Manager` class does not need this data field.

Of course, a realistic program would have many more data fields and methods in each class, but this is enough to demonstrate the power and usefulness of inheritance. The `Worker` class looks like this:

```
public class Worker
{
    int    id;
    float  salary;
    boolean getsOvertime;

    void printCheck()
    {
        // Lots of intricate
        // check-printing code
        // goes here.
    }
}
```

And the `Manager` class looks like this:

```
public class Manager
{
    int    id;
    float  salary;
    Worker[] workers;

    void printCheck()
    {
        // Same intricate
        // check-printing code
        // goes here.
    }
}
```

Despite their differences, these classes have a lot in common. The most worrisome common feature is the `printCheck()` method.

Note Notice the empty parentheses after the method name. This is a common practice when writing about a method. It specifies that you're talking about a method rather than a variable or a class.

`printcheck()` is worrisome because it appears in identical forms in two places. Duplication of code should be avoided, because code is never frozen in time. Code evolves. Over the lifetime of a program, bugs are found and new features are required. The process of fixing bugs and adding features is called *maintenance*, and every program requires it. If a method appears in identical forms in two places, every change must be made twice, and the risk of introducing errors rises dramatically.

It is not surprising that workers and managers share some common features. They are both categories of employees. And here we find a simple but profound truth about the way we humans observe our world.

The [previous chapter](#) presented classes as programmatic representations of mental categories, such as "triangle" or "dog."

Object-oriented programming is a very human approach to writing software, because our minds are good at creating categories for the things we experience in daily life. No doubt all animal species do this to some extent, with categories like "food" and "threat" and "safe place to sleep." People do it best of all.

People are so good at creating mental categories that we take the process one step further. We don't just imagine categories of things. With our talent for abstract thinking, we can imagine categories of *categories*! So the "triangle" category is one member of a larger mental concept that we might call "shapes." Other members of this supercategory are "squares" and "rectangles." Similarly, the "dog" category belongs to the supercategory "mammals," which in turn belongs to its own supercategory: "animals."

The Swedish philosopher Carl Linnaeus organized all living species into a hierarchy of supercategories with seven levels. This organization is still in use among biologists. If you've ever had to memorize "kingdom, phylum, class, family, order, genus, species" for a biology class, you were studying Linnaeus' hierarchy. His structure was more detailed than our "animal, mammal, dog" hierarchy. You can't really say that either hierarchy is more or less correct, though. Each one is appropriate for certain tasks.

Well, enough philosophy. The point is that it's natural to think about hierarchies of categories, and Java supports this way of thinking. Let's see how this is done.

Inheritance from Superclasses

In Java, a category is represented by a class. A *supercategory* (if you will continue to permit the use of this made-up word) is represented by a *superclass*. *Superclass* is a real word, and so is its opposite: *subclass*. Every class can have one superclass. That superclass in turn can have its own superclass, and so on. A class may not have multiple superclasses, but multiple subclasses are allowed.

The `extends` keyword is used to denote the superclass/subclass relationship. To see how this works, let's continue the personnel example from the [previous chapter](#). Right before the philosophical digression, you learned that workers and managers are both categories of employees. You will now create an `Employee` class that will contain all the shared functionality of workers and managers.

In Java, every class is capable of being a superclass, and you don't have to do anything special in the class definition of a class that will have subclasses. (The special work, as you'll soon see, comes when you define the subclasses.) So the superclass looks like this:

```
public class Employee
{
    int    id;
    float  salary;

    void printCheck()
    {
        // The same intricate
        // check-printing code
        // goes here.
    }
}
```

Now let's create the `Worker` subclass:

```
public class Worker extends Employee
{
    boolean getsOvertime;
}
```

The class name is followed by the `extends` keyword, which is followed by the class's superclass. That's all we need to do! This works because in Java, there are two ways for a class to have a variable or method:

- The variable or method can be defined in the class.
- The variable or method can be defined in the class's superclass.

This very simple `Worker` class just defines a single variable. But its superclass (`Employee`) defines the variables `id` and `salary`, as well as the method `printCheck()`, so `Worker` also has those variables and that method. We say that `Worker` *inherits* `id`, `salary`, and `printCheck()` from its superclass.

The `Manager` class is also simple:

```
public class Manager extends Employee
{
    Worker[] workers;
}
```

Again, `Manager` *inherits* `id`, `salary`, and `printCheck()` from its superclass. The situation is diagrammed in [Figure 8.1](#).

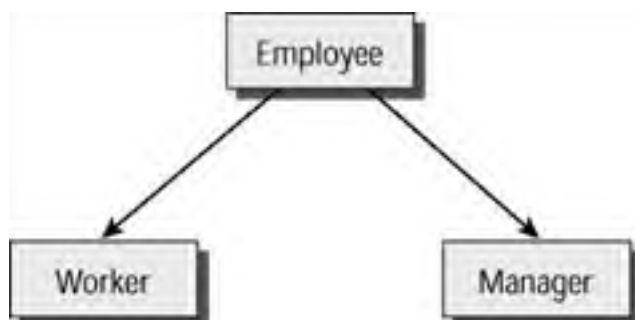


Figure 8.1: A Simple inheritance hierarchy

Figure 8.1 shows that `Employee` is the superclass of both `Worker` and `Manager`. `Employee` itself does not seem to have a superclass, but in Java, every class you define has a superclass, even if you don't explicitly declare one with the `extends` keyword. Java provides a class named `Object`, which is the ultimate ancestor of every class. A class that does not explicitly extend something else extends `Object`.

The Inherit Lab animated illustration lets you create your own class hierarchy diagrams, so that you can see how variables and methods are inherited. To run the program, type `java inherit.InheritLab`. You will see a display that shows a three-level class hierarchy, as shown in **Figure 8.2**.

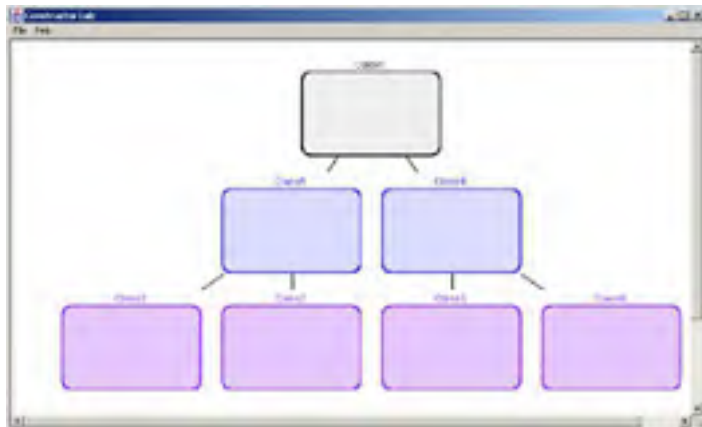


Figure 8.2: Inherit Lab

At the top of the diagram is the `Object` class. `Object` has two subclasses, called `Class1` and `Class4`. Each of those classes has two subclasses. The classes are color-coded based on their level in the hierarchy.

At first the classes are boring. Their names don't mean anything, and they don't have any data or methods. But if you left-click on any class, you'll get a pop-up menu that lets you add a subclass, delete the class, or edit the class. (You can't delete or edit `Object`, since its definition is beyond your control.) First, try adding and deleting classes. Then try editing a class. When you select `Edit` in the pop-up menu, you get a dialog box that lets you change the name of the class, or add or delete data and methods. The dialog box is shown in **Figure 8.3**.



Figure 8.3: Inherit Lab's class-editing dialog box

Try adding a variable to one of the classes in the blue level, just below `Object`. Type a name into the `Add Data` text field, and

then click the Add Data button. Then click Apply. The edit dialog box will go away so that you can see the inheritance diagram. The variable you've added will be seen in the box for the class you edited, and also in all of that class's subclasses, illustrating inheritance of data. You can do the same with methods. Notice that data and methods are color-coded to tell you which class they were defined in.

In the File menu, click on Scenarios. Then look at the two canned hierarchies, which represent animals and transportation. In the Transportation scenario, the bottom-level classes (Car, Bicycle, etc.) inherit from two levels of superclass, as well as from Object. Sophisticated object-oriented programs can have fairly deep hierarchies.

In each scenario, add a subclass at the bottom level and observe the inherited data and methods. Try creating a hierarchy from scratch. If you create something interesting, send us a screenshot or a verbal description at GroundUpJava@sgsware.com. We might include it in the next edition. If so, we'll give you credit.

An Inheritance Example

Let's look at an example of inheritance, expanding on the Worker class from the [previous section](#). Worker is a subclass of Employee, which looks like this:

```
public class Employee
{
    int    id;
    float  salary;

    void printCheck()
    {
        // Whatever.
    }
}
```

Let's add a slightly expanded Worker subclass:

```
1. public class Worker extends Employee
2. {
3.     boolean  getsOvertime;
4.
5.     void dumpSalary()
6.     {
7.         System.out.println("Salary = " + salary);
8.     }
9.
10.    public static void main(String[] args)
11.    {
12.        Worker dagwood = new Worker();
13.        dagwood.salary = 44444.44f;
14.        dagwood.dumpSalary();
15.        dagwood.printCheck();
16.    }
17. }
```

Line 7 of the dumpSalary() method and line 13 of the main() method both act as if salary were an ordinary variable of the Worker class... and they're right. The inherited variables of a class are just like its declared variables. The same is true for inherited methods. Line 15 calls dagwood's printCheck() method, which is inherited.

We will return to this example later on in this chapter. First, it's time to learn what really happens when, as on line 12, an object is constructed.

Construction and Constructors

When you invoke `new` to construct a new instance of a class, automatically a call is made to a piece of code in that class, called its *constructor*. This may come as a surprise, because for the past two chapters you have constructed objects without ever writing constructors, or even knowing about them. In a moment you will see how this works out.

A constructor looks like a method. In fact, there are only two differences between a constructor and a method:

- A constructor has no return type.
- The constructor's name is the same as the class's name.

Like a method, a constructor has a body, enclosed in curly brackets. The code in the body can initialize the newborn object. In fact, this initialization is the basic job of constructors. A constructor can always assume that the object's variables and methods (whether declared in the class or inherited) exist and are accessible for reading, writing, and calling.

Let's add a constructor to the `Worker` class. Assume that all workers in the company have the same salary: \$34,567.89. The `Worker` class (with a different `main()` method) becomes the following:

```
1. public class Worker extends Employee
2. {
3.     boolean  getsOvertime;
4.
5.     Worker()
6.     {
7.         salary = 34567.89f;
8.     }
9.
10.    void dumpSalary ()
11.    {
12.        System.out.println("Salary = " + salary);
13.    }
14.
15.    public static void main(String[] args)
16.    {
17.        Worker dagwood = new Worker();
18.        dagwood.dumpSalary();
19.    }
20. }
```

The constructor is lines 5-8. Constructors, variables, and methods can appear in any order in the body of a class definition. However, it is common practice to have variables come first, followed by constructors, followed by methods. Usually, the `main()` method comes at the end. When this application is run, line 17 constructs a new instance of `Worker`. First, space for all variables is allocated. Then the constructor code is called. Line 7 of the constructor initializes `salary` to 34567.89, so the call to `printSalarydumpSalary()` prints out `Salary = 34567.89`.

Overloading Constructors

It is not especially realistic to expect every worker in a company to have the same salary. Fortunately, you can pass arguments into a constructor the same way you pass them into a method. As with a method, you can put an argument list inside the parentheses that follow the constructor name. Those arguments are accessible within the method. The next version of `Worker` has a constructor that accepts an argument.

```
1. public class Worker extends Employee
2. {
3.     boolean  getsOvertime;
4.
5.     Worker(float sal)
6.     {
7.         salary = sal;
8.     }
9.
10.    void dumpSalary ()
11.    {
12.        System.out.println("Salary = " + salary);
13.    }
14.
15.    public static void main(String[] args)
16.    {
17.        Worker dagwood = new Worker(55555.55f);
18.        dagwood.dumpSalary();
19.    }
20. }
```

The constructor now takes a float argument, and the invocation on line 17 passes a float. The output is `Salary = 34567.89`.

In [Chapter 4, "Methods,"](#) you learned that methods are polymorphic. That is, different methods within a class may share a common name, as long as their argument lists are different. The practice of reusing a method name in a class is called *overloading*. (The term has a negative connotation in real life. When people or bridges are overloaded, that's bad. But in programming, there is nothing bad about overloading.) You can also overload a class's constructor so that the class has multiple constructors, as shown here:

```
public class Manager extends Employee
{
    Worker[] workers;

    Manager(int nWorkers)
    {
        workers = new Worker[nWorkers];
    }

    Manager(float sal, int nWorkers)
    {
        salary = sal;
        workers = new Worker[nWorkers];
    }
}
```

This class has two constructors. In both versions, you specify the number of workers. In the second version, you also specify the manager's salary.

Default Constructors

This section answers an important question: In the [previous chapter](#) and the first part of this one, how was it possible to construct objects in classes that didn't have any constructors? To understand the answer, you have to know what a *no-args constructor* is. It's just a constructor with an empty argument list.

When you create a class with no constructors, the compiler creates a no-args constructor automatically. A no-args constructor that is created automatically is called a *default constructor*. You only get a default constructor if your class does not explicitly have any constructors. If your class has constructors, no matter how many, no-args or otherwise, no default constructor is created for you.

This mechanism assures that every class has at least one constructor, even if the class was written by someone who has never heard of constructors!

A default constructor does almost nothing. It contains no initialization code, because it contains no code at all. All it does is participate in the constructor chain mechanism, which is discussed in the [next section](#).

The Chain of Constructions

Objects are like onions. They consist of layers within layers within layers. Consider the `Submarine` class from InheritLab's Transport scenario. This class extends `WaterTransport`, which extends `Transport`, which extends `Object`. One way to visualize an instance of `Submarine` is as an instance of `Object` forming an inner core. Around this core is a layer consisting of the data and methods of an instance of `Transport`. In turn, this layer is surrounded by a `WaterTransport` layer, which is surrounded by a `Submarine` layer. The layered structure is shown in [Figure 8.4](#).

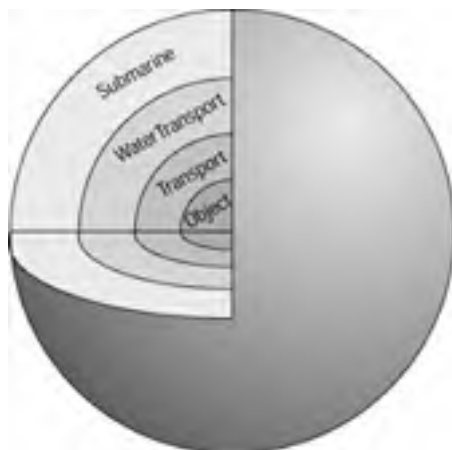


Figure 8.4: Object layers

When a `Submarine` instance is constructed, each layer's constructor is called in turn, starting with `Object` (the innermost layer) and moving outward. This mechanism doesn't have an official name, but we'll call it the *chain of constructors*.

The ConstructorLab animated illustration shows the chain of constructors in action. Start the program by typing `java inherit.ConstructorLab`. At first glance, the program looks just like the InheritLab program that you already saw earlier in this chapter. But when you click on an object, the pop-up menu has an extra Construct... item. If you make this selection, you see an animation of the layer-by-layer construction of the class you've selected. Try it with the `Submarine` class from the Transport scenario.

Here's how the chain-of-constructors mechanism works: When a constructor is called, but before any of its code is executed, a call is made to the no-args constructor of the superclass. If the superclass is not `Object`, before any of its own constructor code is executed, a call is made to its own superclass's no-args constructor. This chain of calls continues up the inheritance hierarchy until it reaches `Object`, which has no superclass.

So when you call a constructor for `Submarine`, the first thing that happens is a call to the no-args constructor for `WaterTransport`. Within that constructor, the first thing that happens is a call to the no-args constructor for `Transport`. Finally, within `Transport`'s no-args constructor, a call is made to the `Object` no-args constructor. At this point, the chain ends.

Why does Java do this? Consider the benefits if you are the person who writes the `Submarine` class code. Your constructors might need to access the data of the `WaterTransport` superclass, which might be initialized by `WaterTransport`'s constructor. That constructor might need to access the data of its own superclass, and so on. The chain-of-constructors mechanism guarantees that by the time a class's constructor code begins to execute, the superclass portion of the class is intact and valid.

Note Bear in mind that the mechanism operates without your having to do anything at all. You don't have to make it happen. As a matter of fact, you can't avoid it. But you can slightly alter its behavior.

As you know, constructor invocation begins with an automatic call to the superclass's no-args constructor. Recall that constructors can be overloaded so that alternate superclass constructors could be available. There are two reasons why you might want to invoke a different superclass constructor:

There is no superclass no-args constructor. This happens if you explicitly give the superclass one or more constructors that take arguments. Now you don't get an automatic default constructor, so unless you explicitly coded a no-args constructor for the superclass, there won't be one.

The superclass has a no-args constructor that doesn't do what you want. But there's another constructor that does exactly what you want.

In either of these situations, you still want the construction chain to happen. You just want to invoke a different version of the superclass constructor. This is done with the `super` keyword.

To see how `super` works, let's extend the `Manager` class from earlier in this chapter. The class looks like this:

```
public class Manager extends Employee
{
    Worker[] workers;

    Manager(int nWorkers)
    {
        workers = new Worker[nWorkers];
    }

    Manager(float sal, int nWorkers)
    {
        salary = sal;
        workers = new Worker[nWorkers];
    }
}
```

This class provides its own constructors, so there is no default constructor provided by the compiler. Neither is there an explicitly coded no-args constructor, so any subclass of `Manager` will have to modify the construction chain to avoid invocation of a constructor that doesn't exist.

Let's create a subclass called `Officer`. An officer is a high-ranking manager who may or may not serve on the board of directors. The subclass will have an `int` variable called `nYrsOnBoard`, which tells how many years (if any) this officer has served on the board. There will also be a single constructor whose arguments are the number of workers reporting to this officer and the initial value for `nYrsOnBoard`. Officer salaries are \$850,000.00. Nice work if you can get it. The `Officer` code looks like this:

```
1. public class Officer extends Manager
2. {
3.     int nYrsOnBoard;
4.
5.     Officer(int nWorkers, int initialNYrs)
6.     {
7.         super(850000f, nWorkers);
8.         nYrsOnBoard = initialNYrs;
9.     }
10. }
```

The line to notice is line 7, which introduces the `super` keyword. It looks like a call to a method called `super()`. In fact, the code in the parentheses is an argument list, but you aren't allowed to create a method with that name. Instead, `super` is a signal that you are modifying the construction chain by requesting a call to a different superclass constructor (that is, one that isn't the no-args version). This use of `super` may only appear as the first executable code in a constructor (so if you reversed lines 7 and 8, you would get a compiler error).

Line 7 invokes an alternative superclass constructor, but which one? This is determined by the argument list inside the parentheses that follow `super`. Here there are two values: a float followed by an `int`. So the `Manager` constructor that gets invoked will be the version that takes two arguments: a float and `int`. If you look at the `Manager` constructor, you'll see that this corresponds to the second of its constructors.

Overriding

You have already seen method overloading, where a method name is reused in a class definition. Java also supports method *overriding*, which looks like it ought to be similar to overloading because the spellings are so similar. In fact, overriding is indeed a kind of method name reuse. In this case, the reuse is within an inheritance hierarchy.

To continue this ongoing example, the `Officer` class is the bottom member of a three-level hierarchy. This is shown in [Figure 8.5](#).



Figure 8.5: Inheritance of `Officer`

In the original version of `Employee`, you imagined a `printCheck()` method. This method is inherited by `Worker`, `Manager`, `Officer`, and any other subclasses of `Employee`, immediate or indirect, that you might create in the future.

But what happens if a class inherits a method that is not appropriate to that class's operation? For example, `printCheck()` might print checks on a monochrome printer that is loaded with low-cost blank check forms. That's fine for ordinary workers, and even for managers, but officers need to have their checks printed by a special color printer, on high-grade fancy paper. So the inherited version of `printCheck()` just won't do.

The solution is to override `Officer`'s inherited version of `printCheck()`. You do this simply by putting into the `Officer` code a version of `printCheck()` that does what you want it to do. The code looks like this:

```
1. public class Officer extends Manager
2. {
3.     int nYrsOnBoard;
4.
5.     Officer(int nWorkers, int initialNYrs)
6.     {
7.         super(850000f, nWorkers);
8.         nYrsOnBoard = initialNYrs;
9.     }
10.
11.    void printCheck()
12.    {
13.        // Whatever, and make it fancy.
14.    }
15.
16.    public static void main(String[] args)
17.    {
18.        Officer julius = new Officer(25, 50);
19.        julius.printCheck(); // Fancy
20.        Worker dagwood = new Worker(44444.44f);
21.        dagwood.printCheck(); // Plain
22.    }
23. }
```

Again, we've left out the details of how to print a check. The import point is line 11, where you have a declaration of `printCheck()` that looks just like the version in `Employee`. The declarations are the same, but the bodies are different. Look at the `main()` code. On line 19, you call `printCheck()` on an officer. The overriding (fancy) version of the method will be called. On line 21, you call `printCheck()` on a worker. Since `Worker` does not override the method, the inherited (plain) version is called.

In order for one method to override another, the superclass version and the subclass version must have identical return types, method names, and argument list types. It is illegal for the return types to be different if the method names and argument list types match. If the names or argument lists are different, that's legal but it isn't overriding.

Overriding allows you to use a very powerful kind of polymorphism, which will be explained in the [next section](#).

Polymorphism Revisited

Recall from [Chapter 4](#) that polymorphism means "many forms," implying "one name, many forms." In [Chapter 4](#), you saw polymorphism involving method overloading (reusing a method name within a class). The [previous section](#) presented overriding, which is polymorphism of a different kind: *name reuse in an inheritance hierarchy*.

This section will present a powerful technique involving overriding polymorphism. Let's begin by exploring the difference between the class of an object and the type of a reference.

The only way to create an object is to call a constructor. Not surprisingly, an object's *class* is the class whose constructor was called when the object was created. So `new Officer(50, 3);` creates an object whose class is `Officer`.

A reference is not an object. You have already seen that a reference is a configuration of 32 bits that uniquely identifies an object. References, not objects, are passed as method arguments. References, not objects, are declared as variables. The *type* of a reference variable is the type that appears in the variable's declaration. So `Worker dagwood;` declares that `dagwood` is a reference of type `Worker`.

So far, this should be glaringly obvious. Almost always, when a reference points to an object, the type of the reference is the same as the class of the object. This happens, for example, in the line `Worker dagwood = new Worker(22222.22f);`.

But sometimes the reference type and the object class are not the same. Let's introduce this idea with an analogy to something you already know about. In [Chapter 3, "Operations,"](#) you learned that you can assign a numeric value to a variable whose type is the same as, or wider than, the type of the numeric value. So you can assign a byte to a float, or an int to a double, as shown here:

```
byte b = 12;
float f = b;
int i = 54321;
double d = i;
```

The new type (on the lhs of the assignment) must have enough capacity to encompass the value. Any extra capacity is no problem. The same rule holds when you're passing an argument to a method. If the method expects an argument of a certain type, you can pass data of any type, provided the type declared in the method is the same as, or wider than, the type you actually pass.

A similar principle applies when you're assigning references. Consider this code:

```
newRef = oldRef;
```

It is legal for `newRef` and `oldRef` to have different types, as long as the type of `newRef` is above the type of `oldRef` in the inheritance hierarchy. So the following is legal:

```
1. Worker dagwood;
2. Employee emp;
3.
4. dagwood = new Worker(22222.22f);
5. emp = dagwood;
```

Here you have one object with two references. Clearly the class of the object is `Worker`, because it is `Worker`'s constructor that is called on line 4. On line 5, the type of reference `emp` is `Employee`, which is the immediate superclass of reference `dagwood`. Thus, the rule is obeyed, and line 5 is legal. You could also pass `dagwood` as an argument to a method that declared it took an `Employee` argument.

So now you know that the type of a reference can be different from the class of the object the reference points to. This puts a subtle but very important restriction on the Java compiler. You and I can look at lines 4 and 5 and say to ourselves, I know `emp` has type `Employee`, but really it points to a `Worker`. *We* can do that, but the *compiler* can't.

This isn't a shortcoming on the part of the people who wrote the compiler. In fact, the developers of the Java compiler are some of the smartest programmers in the world. But there are fundamental theoretical limits on what a compiler can do. No car designer, no matter how brilliant, can make a race car that goes faster than light. Similarly, nobody can create a compiler that, in the general case, can look at a reference and know what the class of the reference's target will be when the code is executed.

To put this more succinctly: *At compile time, the compiler only knows the types of references. It does not know the classes of objects.* This means that a reference's type dictates the variables and methods you can access via that reference. You may only access variables and methods that are the same type as the reference. To return to our example, the reference `emp` has type `Employee`, so by using `emp`, you can read and write variables and call methods of `Employee`. (The data and methods can be implemented in the `Employee` class, or they can be inherited from a superclass.) Using the reference `dagwood` (whose type is `Worker`), you can access the data and methods (whether directly implemented or inherited) of `Worker`. This is shown in [Table 8.1](#):

Table 8.1: References, Variables, and Methods

Variables via emp	Variables via dagwood	Methods via emp	methods via dagwood
Id	Id	printCheck()	printCheck()
salary	salary		dumpSalary()
	getsOvertime		

Given the information in [Table 8.1](#), the previous code example might be baffling. Here is the code:

```
1. Worker dagwood;
2. Employee emp;
3.
4. dagwood = new Worker(22222.22f);
5. emp = dagwood;
```

The table clearly shows that the reference `dagwood` gives you access to all the data and variables you can get to via `emp`, and more. Even though line 5 is legal, why would you ever do it? Line 5 just trades a perfectly good reference for one that is less powerful. There must be some compensating benefit to doing this, or nobody in their right mind would ever want to.

In fact, there is a very valuable compensating benefit, as you will see in the [next section](#).

Inheritance Polymorphism

Let's continue our code example just a bit further. No doubt, there must be a piece of code somewhere in the program that periodically prints a paycheck for everybody who works at the company. Let's suppose there's a class called `Paymaster` that knows who all the employees are. `Paymaster` might look something like this:

```
public class Paymaster
{
    Worker[]    workers;
    Manager[]   managers;
    Officer[]   officers;

    void payEveryone()
    {
        for (int i=0; i<workers.length; i++)
        {
            Worker wor = workers[i];
            wor.printCheck();
        }
        for (int i=0; i<managers.length; i++)
        {
            Manager man = managers[i];
            man.printCheck();
        }
        for (int i=0; i<workers.length; i++)
        {
            Officer off = officers[i];
            off.printCheck();
        }
    }
}
```

In reality, the `Paymaster` class would need a lot more code, including a constructor to set up the three arrays. In fact, there would be a lot more arrays. Companies don't just have workers, managers, and officers. They have presidents, vice presidents, directors, part-timers, and possibly many others. There could be lots of categories of people who need to get paid, and if there had to be one array for each category, that would make for a lot of arrays.

Just to hammer the point home, let's suppose there are classes called `President`, `VP`, `Director`, and `PartTimer`, each of which extends `Employee`. We won't show the code for these classes, but here is the monster that `Paymaster` has become:

```
public class Paymaster
{
    Worker[]    workers;
    Manager[]   managers;
    Officer[]   officers;
    President   prez;    // No array: there's only 1 president
    VP[]        vps;
    Director[]  directors;
    PartTimer[] partTimers;

    void payEveryone()
    {
        for (int i=0; i<workers.length; i++)
        {
            Worker wor = workers[i];
            wor.printCheck();
        }
        for (int i=0; i<managers.length; i++)
        {
            Manager man = managers[i];
            man.printCheck();
        }
        for (int i=0; i<workers.length; i++)
        {
            Officer off = officers[i];
            off.printCheck();
        }
        prez.printCheck();
        for (int i=0; i<vps.length; i++)
        {
            VP veep = vps[i];
            veep.printCheck();
        }
        for (int i=0; i<directors.length; i++)
        {
```

```
        Director dir = directors[i];
        dir.printCheck();
    }
    for (int i=0; i<partTimers.length; i++)
    {
        PartTimer pt = partTimers[i];
        pt.printCheck();
    }
}
}
```

Let's see how much you can simplify this code, using inheritance polymorphism. The first thing to do is eliminate all those arrays and replace them with a single array, called `employees`:

```
public class Paymaster
{
    Employee[] employees;
    . . .
```

The components of the new array aren't really employees. That is, `employees` is an array of references whose types really are `Employee`, but the classes of the objects pointed to by those references are really `Worker`, `Manager`, `Officer`, and so on. The array is initialized by a lot of code along the following lines, which might appear in `Paymaster`'s constructor:

```
. . .
Worker      dagwood;
Manager     julius;
President    preston;
Director    deirdre, dirwood;

. . .

employees[1154] = dagwood; // Employee <- Worker
employees[1155] = julius;  // Employee <- Manager
employees[1156] = preston; // Employee <- President
employees[1157] = deirdre; // Employee <- Director
employees[1158] = dirwood; // Employee <- Director

. . .
```

The `employees` array is a cluster of references, all of type `Employee`. Each of the 5 commented assignment lines stores a reference in a component of the array, and not one of those references is actually of type `Employee`. That's okay. The rhs references are all of types that are subclasses of `Employee`, so the "up-the-inheritance-hierarchy" assignment rule is obeyed.

Now let's return to `Paymaster`'s `payEveryone()` method. Here is all you have to do:

```
void payEveryone()
{
    for (int i=0; i<employees.length; i++)
    {
        Employee emp = employees[i];
        emp.printCheck();
    }
}
```

That's all! All the references to all the people are now living peacefully together in one diverse community... er, array of references, where the classes of the objects pointed to are unknown and mixed. But you do know that every class is a subclass of `Employee` (or is `Employee` itself). So every object has a `printCheck()` method. This method might be the version inherited from `Employee`, or it might be an overriding version.

What happens when the `payEveryone()` loop pays an officer? Recall that the `Officer` class overrides `printCheck()` to use a fancy printer with fancy paper. You have a reference (some component of the `employees` array) of type `Employee`, pointing to an object of class `Officer`. Each has its own version of `printCheck()`. Which one wins?

The answer, and this is crucially important, is that *the type of the reference is ignored*. The class of the object being called determines which version of an overridden method will be called. So in this example, all the officers will get their checks printed in fancy paper, and any other classes that override `printCheck()` will have the appropriate version called.

In case this is overwhelming you, let's look at a very simple example that illustrates the same principle:

```
public class FlyingMachine
{
    void whoAreYou()
    {
        System.out.println("I am a flying machine.");
    }
}
```

Subclass `FlyingMachine` like this:


```
public class Helicopter extends FlyingMachine
{
    void whoAreYou()
    {
        System.out.println("I am a helicopter.");
    }

    public static void main(String[] args)
    {
        FlyingMachine fm = new Helicopter();
        fm.whoAreYou();
    }
}
```

When the application runs, the output is "I am a helicopter." This proves that the class of the object, not the type of the reference, determines the method version that gets called.

Team LIB

← PREVIOUS NEXT →

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Which of the following hierarchies illustrate a good understanding of the difference between classes and objects? Which ones represent mistaken understanding? The arrows mean "has subclass", so in option A, Shape → Triangle means "class Shape has subclass Triangle".

1. Shape → Triangle → RightTriangle
2. GreatLiterature → GreatPoem → DivineComedy
3. Planet → Continent
4. Person → HeadOfState → Emperor
5. Person → HeadOfState → Emperor → AugustusCaesar

2. Which of the following classes have a no-args constructor?

1. A)

```
class A { }
```
2. B)

```
class B
{
    B() { }
}
```
3. C)

```
class C
{
    C(int x) { }
}
```
4. D)

```
class D
{
    D(int y) { }
    D() { }
}
```

3. Write the code for two classes. The first, called `WaterBird`, has a float variable called `weight`. The class has a single constructor that looks like this:

```
WaterBird(float w)
{
    weight = w;
}
```

Compile this class. Now create the second class, called `Duck`, which extends `WaterBird`. `Duck` has no variables or methods, so it shouldn't take you long to write it. Will `Duck` compile? First, think about the issues involved. Then try to compile `Duck` and see if you were right.

4. Write some code to demonstrate to yourself the chain of construction. Create an inheritance hierarchy of 4 classes. Give them any names you like. They don't have to have any data or methods, but each one should have a no-args constructor. These constructors should print out a line identifying the current class (something like "Constructing an instance of `WaterBird`"). Your `main()` method should construct a single instance of your lowest-level subclass. What is the output? Does it matter which class contains the `main()` method?
5. Write some code to demonstrate inheritance polymorphism. Create a superclass class with 3 subclasses. The superclass should have a method that prints out a line identifying the current class (something like "I am a Monster"). Two of the subclasses should override this method to print out a different message (like "I am a Werewolf"). Give the superclass a `main()` method with an array of size 4, typed as the superclass (for example, `Monster[] monsters = new Monster[4];`). Your `main()` should populate the array with references to 4 objects, each with a different class, and then traverse the array, calling your method on each array component. What is the output? Does it matter which class contains the `main()` method?

Chapter 9: Packages and Access

Overview

Congratulations! At this point, you know almost all there is to know about Java's classes. The remainder of this book will look at how classes interact, and it will present many of the core Java classes that the system provides to make your life easier. To make an analogy with the life sciences, we are pretty much done with class anatomy (the analysis of the internal structure of a class) and are ready to tackle class sociology (the study of how classes interact).

This chapter will first look at packages, which are organizations of interrelated classes. Once you understand packages, you will have a good foundation for understanding access. Java has several keywords that control access, including `public`. This means that by the time you finish this chapter, you will know why your application classes and your main methods have been marked as `public`.

By the way, in this edition of this book, this chapter has no animated illustrations. The information presented here doesn't benefit from the animated illustration model. However, if you can think of a concept from this chapter that would look good as an animation, please send us your idea in detail at groundupjava@squares.com. If we use your idea in the next edition, we will give you a credit.

Packages

A *package* is a named group of classes. Generally, the classes of a package are collected together into a directory. It is possible for package classes to appear in more than one directory, but it's hard to imagine when this would be helpful. So a package looks a lot like a directory, even though they aren't exactly the same thing.

There is another similarity: Just as a directory can contain files and subdirectories, a package can contain classes and subpackages. For example, a package called `acmeproducts` might contain two classes named `Database` and `Connection`. The package might also contain a subpackage called `utilities`, which contains three classes named `ThreadPool`, `Mailbox`, and `UserProfile`. Your package structure would most likely appear in the directory structure shown in [Figure 9.1](#).

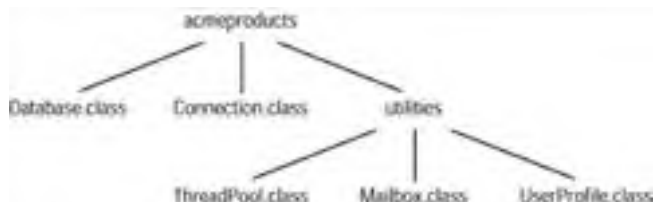


Figure 9.1: Example package/ directory structure

It's important to realize that the directory and the subdirectory shown in [Figure 9.1](#) are not the actual package and subpackage. They are just the places where the classes of the package and subpackage are found. Soon you'll learn how to put your own classes in packages, and why you might want to do so. For now, be aware that there is more to it than just creating the right directory structure and storing the class files appropriately.

When a class is part of a package, the class has a long, formal, official name. It's important to understand this long name, even though it's rarely used. The official name of a class consists of its package structure, from top to bottom, followed by the class name as defined in the source file. All these elements are separated by periods. For example, the `Mailbox` class in [Figure 9.1](#) would be defined in a file called `Mailbox.java`. Its full name is `acmeproducts.utilities.Mailbox`, because it lives in subpackage `utilities`, which lives in package `acmeproducts`. (Note that the package name is all lowercase. You are allowed to use uppercase in package names, but by convention, nobody does.)

There is yet another parallel between directories and subpackages. Even a modest laptop can have tens of thousands of files on its hard drive. If every file on the drive had to have a unique name, keeping track of which names were still available would be a horrendous task. Thanks to directories, you only have to maintain name uniqueness within directories. So you might have a directory called `photos`, with a subdirectory called `NewYearsParty`, which contains a file called `JulieAndRich.jpg`. If you had a housewarming party, you could create a subdirectory of `photos` called `housewarming`, and if you took a picture of Julie and Rich at the housewarming, you could store it in the `housewarming` subdirectory under the name `JulieAndRich.jpg`.

We say that a directory structure provides a *namespace*. A namespace is a way of organizing resources (files, classes, etc.) so that name uniqueness only has to be maintained in relatively small and manageable regions.

Packages are also namespaces. Within a package, all class names must be unique. However, names may be reused in different packages without restriction. [Figure 9.2](#) shows a package structure that might be used by a fictional company called Stained Glass Software. This company has two product lines: database products and ray-tracing software.

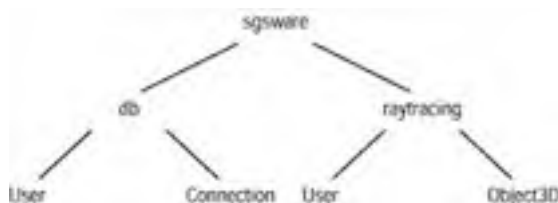


Figure 9.2: Package as namespace

Stained Glass Software has dozens of programmers, working in two divisions on opposite sides of the world. Life would be impossible if every programmer had to check with every other programmer before creating a new class, just to make sure the class name wasn't already in use. As you can see from [Figure 9.2](#), both divisions have created a class called `User`. Fortunately this isn't a problem, because the two classes are in different packages, and packages are namespaces. To put this another way, one class is really called `sgsware.db.User`, and the other is really called `sgsware.raytracing.User`. You can see that packages support collision-safe naming. In the real world, a package might be developed and maintained by several workgroups, by a single workgroup, by a few individuals, or by a single person.

Creating Your Own Package

By now, we hope you're convinced that packages are a good thing. Here's how to create your own packaged classes.

You need to do two things:

- Use the `package` keyword in your class source code.
- Compile with the `-d` flag.

It's interesting to think about what *isn't* in this list. Here's what you *don't* have to do:

- Create a package.
- Create the package directory structure.
- Move the class files into the directory structure.

Let's suppose you are the founder of Stained Glass Software. You have a new computer, fresh out of the box, and you are ready to write the `sgsware.raytracing.Object3D` class. This is the company's first class, so no structure has been created yet.

The first step is to create a directory where the package structure will go. Let's say you decide to put it in `/products/revA`. (You might be using the kind of computer that uses the backslash as a file path separator, but for simplicity we'll use forward slashes throughout this book.) After making sure that `/products/revA` exists, you create a directory to hold your source code. We'll call it `/products/source/ray`. So initially, your directory structure looks like [Figure 9.3](#).

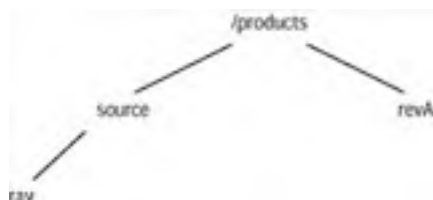


Figure 9.3: Initial directory structure

When you write your source code, you have to tell the compiler that the class belongs to the `sgsware.raytracing` package. To do this, you use the `package` keyword. Besides comments, the package declaration must be the first code in your source file:

```
// This class belongs to a package.
package sgsware.raytracing;

public class Object3D
{
    . . .
}
```

When you compile, use the `-d` command line option. This should come after `javac`, and it should be followed by a space. After the space comes the directory where the package structure is to be stored. Since you're putting your package of classes in `/products/revA` (known as the *destination directory*), you would compile like this:

```
javac -d /products/revA Object3D.java
```

The destination directory must exist before the command is typed. The compiler realizes that the class file should be `/products/revA/sgsware/raytracing/Object3D`. The compiler will create any required subdirectories in the destination directory, and it will place the class file it generates in the appropriate place. So after compilation, your directory structure would look like the one shown in [Figure 9.4](#).

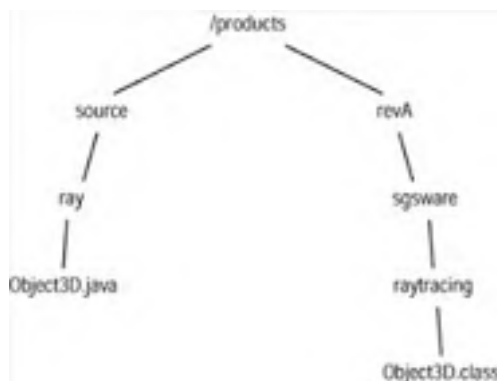


Figure 9.4: After compilation

When you want to create a second `raytracing` class, you can put your source file anywhere you like. However, if you put the source file anywhere other than `/products/source/ray`, it will only make life more baffling for yourself and others. Again, your source should declare that it belongs to the `sgsware.raytracing` package:

```
package sgsware.raytracing;

public class User
{
    . . .
}
```

When you compile, you again use the `-d` flag:

```
javac -d /products/revA User.java
```

This time, the compiler does not have to create a subdirectory for storing the generated class file, since that subdirectory already exists. After compilation, your directory structure looks like [Figure 9.5](#).



Figure 9.5: After more compilation

Now that you know how to create your own packages, let's look at how to use them.

Finding Packages: *classpath*

The designers of Java assumed that you would be working in an environment where you would be using lots of different packages. This was a safe assumption, since Java itself comes with lots of different packages that support functions like string manipulation, file I/O, and GUI components. Moreover, you are likely to be using other packages that you've created yourself, that are standard for your company, or that were bought from a third party. You might be developing code that uses classes from many different packages. When you compile and execute, the compiler and JVM need to know where in your file system these custom packages are located. You do this with the *classpath*.

The classpath is a list of directories, or *classpath elements*, that contain package structures. The classpath elements can be specified in two places:

- The CLASSPATH environment variable
- The -classpath or -cp argument of the javac or java command line

An environment variable is a variable whose scope is your computing session, rather than the interior of a program. Individual programs can read environment variables and take action accordingly. The CLASSPATH variable, which is read by both the Java compiler and JVM programs, is a list of classpath elements, separated by semicolons (;) for Windows machines and by colons (:) for other systems.

You can set CLASSPATH either by typing a command or by running a script. Running a script is easier (after you create the script). Some people prefer to set CLASSPATH in their boot or login scripts (or whatever the equivalent is on their own machines). Appendix A, "Installing Java" shows how to write a script that sets CLASSPATH to ".", which is the current working directory. Different operating systems use different commands to set an environment variable, as detailed in the appendix.

The other way to specify classpath elements is to type them into your compilation and execution command lines. You do this after the javac or java command: type `-classpath`, then a space, then the classpath elements you want to specify. As with CLASSPATH, if you have more than one element, they should be separated by semicolons (;) for Windows machines and by colons (:) for other kinds of machines.

Suppose you are using a Windows machine, and you have acquired and built lots of custom packages. These packages are stored in three different directories: `\a\b`, `\c\d`, and `\e\f`. There might be more than one package in any of these directories. The names aren't very creative, and three is an inconveniently large number of classpath elements, but it makes for a nice clear example.

Now suppose that, for some reason, you want to specify `\a\b` and `\c\d` in CLASSPATH, while specifying `\e\f` on the command line. You would start by setting CLASSPATH (either manually or in a script) as follows:

```
SET CLASSPATH=\a\b;\c\d
```

Then you would compile source like this:

```
javac -d \my\destination -classpath \e\f MyThing.java
```

If MyThing is your application class, and it lives in package `sgsware.db`, you would execute your application like this:

```
java -classpath \my\destination;\e\f sgsware.db.MyThing
```

The application class, as well as any other classes used at any time in the application, must appear in one of the classpath elements. So the MyThing.class file must be in one of the following directories:

- `\a\b\sgsware\db`
- `\c\d\sgsware\db`
- `\e\f\sgsware\db`

Note that each of these directories consists of a classpath element, followed by a package structure.

The Java compiler and the JVM use a piece of code called a *class loader*. The class loader finds class files, reads them, and translates them into internal representations. The first step in the process is to find files. The class loader does this by looking in each classpath directory in turn. In each directory, it looks for a subdir that corresponds to the package of the class being loaded. So when the class loader looks for MyThing.class, it looks in each of the directories listed in the preceding bulleted list.

Now you know how to create, store, and use packages. But there is still a problem, which will be presented and solved in the [next section](#).

Importing

You have seen that a class name is really a list of package elements, separated by periods and ending with the (short) name of the class. Packages provide a convenient way to organize software and reduce naming headaches, but there seems to be a tradeoff with what happens in your source code.

In the [previous section](#), you considered a class called `MyThing` in a package called `sgsware.db`. The true name of class `MyThing` is `sgsware.db.MyThing`. That doesn't seem so bad until you realize that the following line of source code is not allowed:

```
MyThing m = new MyThing();
```

This line won't compile, because it doesn't use the true name of the class. The line ought to be

```
sgsware.db.MyThing m = new sgsware.db.MyThing();
```

Programming wouldn't be very much fun if you had to use full class names everywhere. Imagine what a burden it would be, if you had to do so much typing. You would soon find yourself wishing for a way to abbreviate: "I wish I could tell the compiler that every time I type `MyThing`, I really mean `sgsware.db.MyThing`." This wish is granted by Java's import feature.

Importing is a very useful feature with an unusual name. The name comes from earlier object-oriented languages, in which the functionality was a kind of symbolic importation. Now the keyword continues to be used in Java, but the functionality has more to do with abbreviation than with importation. The syntax of `import` is

```
import full.class.Name;
```

You can have as many import statements as you like in a source file. Imports must appear before the class declaration, as shown in the following code. It assumes that you want to use the `Employee` and `Manager` classes of a package called `biz`:

```
1. package sgsware.raytracing;
2. import biz.Employee;
3. import biz.Manager;
4.
5. public class User
6. {
7.     Employee   dagwood;
8.     Manager    dithers;
9.
10.    . . .
11. }
```

Thanks to the imports on lines 2 and 3, you can use abbreviated class names on lines 7 and 8 (and everywhere else in this source file). Without the imports, lines 7 and 8 would have to be

```
7.    biz.Employee   dagwood;
8.    biz.Manager    dithers;
```

Sometimes a source file might need to use many or all of the classes in a large package. It would be cumbersome to type in the names of all those classes, one per import line. In the spirit of supporting abbreviation, you are allowed to use an asterisk (*) in place of a class name. This causes all classes in the package to be imported. So the preceding code could be slightly shortened as follows:

```
1. package sgsware.raytracing;
2. import biz.*;
3.
4. public class User
5. {
6.     Employee   dagwood;
7.     Manager    dithers;
8.
9.     . . .
10. }
```

One last note on importing: A class imports all the other classes in its package automatically. So you never have to do the following:

```
package mypackage;
import mypackage.*;
. . .
```

Now that you understand how packages work, you have a foundation for learning about Java's various access modes. These will be presented in the [next section](#). You also have a basis for understanding Java's core classes. These will be introduced in [Chapter 12, "The Core Java Packages and Classes,"](#) and will be presented throughout the remainder of the book.

Access

Java's access control is based on the idea that certain features of a class should not be usable by other classes. Before you learn the details of access control, let's look at why this idea is sound.

One of the fundamental concepts of object-oriented programming is *data hiding*. This is the practice of making a class's data as inaccessible as possible to other classes. Why would this be beneficial?

Often there are many valid ways to represent information. A temperature might be stored as degrees Kelvin, Celsius, or Fahrenheit. A price might be listed in various currencies. A color might be represented as a name, as red/green/blue levels, as red/yellow/blue levels, or as hue/ saturation/brightness levels. Maintenance considerations might force class code to be rewritten so as to change the internal representation. For example, if an Italian company bought a company in the United States, money representation might be converted from dollars to euros, and temperature representation might be converted from Fahrenheit to Celsius.

Even if data representation does not change, it makes sense to localize the code that knows about representation inside a single class. It is wasteful to make all classes know how data is represented internally, and it creates the risk of bugs (if the other classes misinterpret the internal representation).

Imagine a class called `Thermometer`, which somehow reads a physical thermometer device. A very clean design is to give the class a `getTempCelsius()` method. The method name leaves no room for confusion as to the units of the return value. There could also be `getTempFahrenheit()` and `getTempKelvin()` methods, so that nobody ever has to look up the conversion formulas. Moreover, nobody ever needs to know how temperature is represented within the class. It might be Fahrenheit, Celsius, or Kelvin. It might change from one rev of the class software to another. The benefit to those of us who use the `Thermometer` class is enormous: We never have to worry about the internal representation.

The general principle of data hiding is that an object's data should never be accessed directly from outside the object. Instead, the object's class should provide methods for reading and setting the data. These methods are officially called *accessors* and *mutators*, but they are often called by their nicknames: *getters* and *setters*. An accessor/getter has an empty argument list and returns a data value. A mutator/setter has a void return type and a single argument. By common convention, the name of an accessor method begins with `get`, followed by the property to be retrieved. The name of a mutator begins with `set`, followed by the property to be modified.

To support this data-hiding approach, object-oriented languages provide facilities to let you restrict access to a class's data and methods. In Java, this is done with *access modifier* keywords. Java has three access modifiers:

- `public`
- `private`
- `protected`

These keywords appear before the declarations of the data or methods they apply to. The `public` modifier may also appear before a class definition. Before we define what the various access modes mean, let's look at an example to clarify the syntax:

```
public class AccessExample
{
    public int         x;
    private double    d;
    protected static float f;
    char              c;

    public int getX()
    {
        return x;
    }

    private void printC()
    {
        System.out.println("c = " + c);
    }

    protected void setD(double newD)
    {
        d = newD;
    }

    void bumpX()
    {
        x++;
    }
}
```

Access modifiers cannot apply to data defined within a method. (Since such data ceases to exist after the method returns, we don't need to think about which outside classes may use it.) Notice the declaration of `f`, which is both `protected` and `static`. Access modifiers can be freely combined with non-access modifiers such as *static*. Modifier order is unrestricted, so you could equivalently say `static protected float f`;. For the sake of readability, it's good practice to align the first character of your variable names on a tab stop, as the preceding example shows.

Java has three access modifier keywords, but four access modes. The fourth access mode is what you get if you don't specify `public`, `private`, or `protected`. This fourth mode is called *default*, although you might also sometimes see it called *package* or *friendly*. In the preceding example, variable `c` and method `bumpX()` both have default access.

Now let's look at what the different access modes do.

Public Access

Public access is completely unrestricted. Classes, data, and methods can be designated public. A public class can be used by any other class. Public data can be read and written by any code (violating the spirit of data hiding). Public methods can be called from any code.

When you run an application, the Java Virtual Machine creates a class loader, which loads your application class. The class loader is itself a class, and your application class must be public so that the loader can load it.

Private Access

The most restrictive access mode is private. Data and methods can be designated private, but not classes. (There is a kind of class called an inner class that can be private or protected, but inner classes are beyond the scope of this book.) A private variable can be written or read only by an instance of the class that defines the variable. A private method can be called only by an instance of the class that defines the variable.

Private access may not be quite as private as you expect. Let's look at an example:

```
1. public class Employee
2. {
3.     private float salary;
4.
5.     public float getSalary()
6.     {
7.         return salary;
8.     }
9.
10.    public void setSalary(float newSalary)
11.    {
12.        salary = newSalary;
13.    }
14.
15.    public boolean earnsMoreThan(Employee other)
16.    {
17.        if (salary > other.salary)
18.            return true;
19.        else
20.            return false;
21.    }
22.
23.    . . .
24. }
```

On line 3, `salary` is declared private. This makes sense, because one's salary should be kept private. The `getSalary()` and `setSalary()` methods are in the spirit of data hiding. Public methods get and set private data, and there are no surprises. But look at line 17, where the code compares the current employee object's salary to the salary of a different employee. Any object that executes line 17 reads a private variable of a different object.

That's just how private access works. Any instance of `Employee` can read and write not just its own private data, but the private data of any instance. Similarly, any instance of `Employee` can call any private method on any instance of `Employee`.

Default Access

Default access is all about packages. Fortunately, you have just learned all about packages, so you're in great shape to learn about default access.

Default access does not correspond to a modifier keyword. Instead, it's the access mode you get when you don't mark a class, variable, or method as public, private, or protected. A default-access class can be used by any instance of any class that's in the same package. A default-access variable can be read or written only by an instance of a class in the same package as the class that owns the default-access variable. A default-access method can be called only by an instance of a class in the same package as the class that owns the default-access variable. In other words, anything with default access can be used by anything in the same package, and it cannot be used from outside the package.

Default access is useful in this common situation: You sell a package of classes that solve a particular problem. A few of the classes are for direct use by your customers. These are public and thoroughly documented (you'll see how this is done in [Chapter 11](#)). The rest of the classes in the package perform purely secondary roles. They are never used directly by your customers, and are used only by the public classes to help in their internal workings.

There is no need for your customers to know about these secondary classes. In fact, everybody is better off if nobody but you knows about them. This is true not just for classes, but for data and methods as well. Some classes, data, and methods are part of your package's publicly visible interface, while others are nobody's business but your own.

We can draw a parallel between private features in a class and default-access features in a package. In a class, private data and methods are only for the internal working of the class. In a package, default-access classes, data, and methods are only for the internal working of the package.

You have already seen packages with default-access features, although you may not have realized it. When a JVM is executed, it builds a package called the *unnamed package*. This consists of all classes in the current working directory that do not explicitly contain package declarations. Consider the example classes you used in the [previous chapter](#): `Employee` and its subclasses `Worker`, `Manager`, and so on. Those classes didn't use packages, so the obvious way to proceed would be to put them all in the same directory, and to compile with a command like `javac *.java`. Eventually, all the class files would exist in the current working directory. Assuming one of those classes had a `main()` method, a JVM could run that application. Then all the classes

would be considered to be in the no-name package. None of the data or methods in those classes had access modifiers (since access modifiers weren't introduced until this chapter), so they got default access invisibly. And everything could work smoothly. Any instance of any class could use any data, and call any method, of any instance of any class.

The no-name package allows your code organization to evolve as your understanding becomes more sophisticated. At first, you don't know about packages or access. Related source files live together in a directory, along with the corresponding class files. Everything can access everything. Later on, you learn about packages and access. You have several reasonable choices for organizing your source code. All source files can be together in the same directory, or the source can be organized into subdirectories that reflect the package structure, or you can use some other scheme. No matter how you organize your source code, your class files are organized automatically (by the compiler, when you use the `-d` option) into directories that reflect the package structure.

If you are developing code that involves more than just a few classes, it's a good idea to use packages. For smaller projects, it's fine to use a single directory and take advantage of the no-name package. In this book, the code examples are as simple as possible. Example classes have package declarations only when using a package is relevant to the topic of the example.

Protected Access

Protected access is default access plus a little bit more. Only data and methods may be protected; classes may not. (Actually, inner classes may be protected, but they are beyond the scope of this book.)

Protected access is useful in a certain interesting situation. You already saw that default access comes into play when you are sharing a package of interrelated classes, some of which will be directly used by your customers. Protected access comes into play when you are sharing a package of interrelated classes, some of which will be *subclassed* by your customers.

It makes sense for your customers to leave your package intact. You certainly don't want dozens of different evolutions of your package out there in the world, one evolution per customer. It is cleaner for everyone if the various subclasses created by your various customers are in separate packages. But the subclasses might want access to non-public data or methods of their superclasses. Even if those desirable variables and methods had default access, they still wouldn't be useful because they would be in a different package. So protected access grants access to subclasses of the class that owns the protected features, even if the subclasses live in different packages.

A protected method may be overridden in any subclass of the class that owns the method, even if the subclass is in a different package. A protected method may be called by any instance of any subclass of the class that owns the method, even if the subclass is in a different package.

Protected data is more complicated than protected methods. If a variable is protected, it is *not* accessible by just any instance of any other-package subclass. It can be accessed only by the instance of the other-package subclass that owns the data.

Let's look at some simple examples. Here's a superclass:

```
package mystuff;
public class Fish
{
    protected float weight;
}
```

Here's a subclass *in a different package* that makes appropriate use of the protected variable:

```
import mystuff.Fish;

package yourstuff; // Different package!
public class Tuna extends Fish
{
    void printWeight()
    {
        System.out.println("I weigh: " + weight);
    }
}
```

The code is legal because any instance of `Tuna` that executes `printWeight()` is accessing its own version of the protected variable `weight`.

Now here's an example that won't compile, to show you what protected access does *not* mean:

```
import mystuff.Fish;

package yourstuff; // Different package!
public class Tuna extends Fish
{
    void printSomeonesWeight(Fish someone)
    {
        System.out.println("Someone weighs: " +
                           someone.weight);
    }
}
```

This code is illegal, because protected access doesn't mean that any `Tuna` may access any other `Tuna`'s protected data. Protected access is different from private access in this regard.

Bear in mind that this restrictive meaning of "protected" only matters in a different-package subclass. The following code is perfectly legal:

```
package mystuff; // Same package as superclass
public class Snapper extends Fish
{
    void printSomeonesWeight(Fish someone)
    {
        System.out.println("Someone weighs: " +
                           someone.weight);
    }
}
```

Here the situation is different, because the subclass and the superclass are in the same package. Remember that protected access is default access plus a little more. Even if `weight` were default instead of protected, it could be accessed by any class in the same package, independent of any superclass-subclass relationships.

Access and Overriding

Java has a rule that seems strange at first glance: When you override a method, the subclass's version may not have a more restrictive access mode than the superclass's version. This is shown in [Table 9.1](#):

Table 9.1: Legal Access Modes for Overriding Methods

Superclass Version Access	Subclass Version Access
Public	public
Protected	public, protected
Default	public, protected, default
Private	public, protected, default, private

This rule seems arbitrarily restrictive, but on closer inspection, it is absolutely necessary. The [previous chapter](#) discussed polymorphism, and you saw what a powerful tool it can be. It turns out that polymorphism is only possible if you have the access/overriding rule. Let's see why this is.

[Chapter 8](#) presented the `Employee` class, which had a `printCheck()` method. `Employee` had a subclass called `Manager`, which had a subclass called `Officer`. The `Officer` class overrode `printCheck()`. You saw that you could have an array, typed as `Employee[]`, that contained references to objects of a variety of classes. Either `Employee` itself or any of its subclasses were allowed, as shown in [Figure 9.6](#).

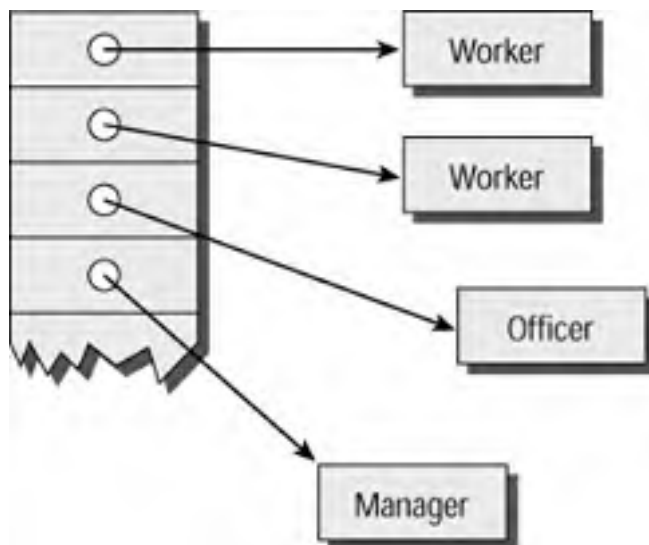


Figure 9.6: Polymorphism revisited

You could then traverse the array as follows:

```
for (int i=0; i<employees.length; i++)
    employees[i].printCheck();
```

Each object would use its own class's version of the `printCheck()` method. This is especially useful if some subclasses override the method.

The clean polymorphic system breaks down if a subclass is allowed to override a method so that the method's access becomes more restricted. To see why this is so, let's assume that the check-printing loop is in some method in a class called `Paymaster`. Let's create a new subclass of `Employee`, called `PartTimer`:

```
class PartTimer extends Employee
{
    private void printCheck()
    {
        // Whatever
    }
}
```

This class won't compile. That's good. If the class were allowed to override `printCheck()` as shown (making its access more restricted, thus violating the rule), it would not be possible for an instance of `Paymaster` to call `printCheck()` on an instance of `PartTimer`. A private method can be called only by an instance of the owning class, so a `PartTimer`'s `printCheck()` could be called only by an instance of `PartTimer`.

Consider what would happen in the absence of this rule. The polymorphic loop in `Paymaster` would work its way through the array, calling `printCheck()` on workers, managers, and officers. Eventually, it would need to make an illegal call to `printCheck()` on a `PartTimer`. This situation must be avoided, and the designers of Java had several options for preventing it. The no-restrictive-overriding rule is sometimes an inconvenience, but actually it is an excellent solution because it gives priority to clean polymorphism.

Team LIB

PREVIOUS NEXT

Final and Abstract

This section will look at two more modifiers: `final` and `abstract`. They aren't access modifiers, but this is still a good place to present them.

Final

Classes, methods, and data may be designated final. A final class may not be subclassed. A final method may not be overridden. A final variable may not be modified after it is initialized.

Final data is useful for providing constants. For example, you might have a `Zebra` class that provides the zebra's weight in pounds or kilograms:

```
class Zebra extends Mammal
{
    private double weightKg;

    public double getWeightKg()
    {
        return weightKg;
    }

    public double getWeightLbs()
    {
        return weightKg * 2.2;
    }
}
```

This class uses appropriate data hiding. A zebra's weight is stored in kilos (the variable name leaves no doubt there), but users of the class never need to know that. Let's assume that eventually the class will have many methods that convert back and forth between kilos and pounds. There will be a lot of multiplying and dividing by 2.2. The standard approach to this situation is to declare a constant:

```
class Zebra extends Mammal
{
    static private final double KGS_TO_LBS = 2.2;

    private double weightKg;

    public double getWeightKg()
    {
        return weightKg;
    }

    public double getWeightLbs()
    {
        return weightKg * KGS_TO_LBS;
    }
}
```

The constant is called `KGS_TO_LBS`. It is static because its value is always going to be the same for all instances of the class, so there is no benefit in giving each instance its own non-static copy. It is private because it is only for use inside the class. It is final because its value should never change under any circumstances. Constants require a little extra typing, but they are well worth the effort for three reasons:

- They explain what they do. Someone reading the code, especially someone who doesn't recognize 2.2 as the kilogram-to-pounds conversion factor, will instantly understand the intention of a constant named `KGS_TO_LBS`.
- They eliminate the need to look up or memorize conversion factors and similar values.
- They provide protection against typos.

The third point requires an example. Suppose you aren't using constants, and it's late at night, and you're tired. Somewhere in the `Zebra` source code, which is now thousands of lines in length, your finger slips and you accidentally type `3.3` instead of `2.2`. It could take a long time for the error to manifest itself, and when it does, you will have to sort through thousands of lines of code to find the problem.

On the other hand, suppose you are committed to using the constant. It is still late at night, and your finger slips, and you accidentally multiply by `KGS_TO_LBX` instead of `KGS_TO_LBS`. The next time you compile your code, the compiler will complain that variable `KGS_TO_LBX` does not exist. When you use constants, the compiler finds your typos for you.

Abstract

Classes and methods may be designated abstract; data may not.

An abstract method has no method body. All the code from the opening curly bracket through the closing curly bracket is gone, replaced with a single semicolon (;). Here is an example of an abstract method:

```
abstract protected double getAverage(double[] values);
```

The `abstract` keyword may be combined with the `public` and `protected` access modifiers. Here you see a method that is both abstract and protected. There is nothing unusual about the declaration part of the method. It only gets strange after the parenthesis that closes the argument list. Where you would expect to find the method body, there is only a semicolon.

When a class has an abstract method, that method's implementation will be found in the class's subclasses. In a moment you'll see an example, but first let's cover a few rules governing abstract classes and methods.

An abstract class may not be instantiated. That is, you are not allowed to call any constructor of any abstract class. Also, if a class contains any abstract methods, the class itself must be abstract. You might say that an abstract class is one that is incomplete: It lacks one or more method implementations.

Suppose you want to create several classes, all of which share some functionality and model similar real-world things. This strongly indicates that the classes should extend a common superclass, which should contain the shared functionality. Every subclass will inherit the common methods, so this is a good object-oriented design. It would not be unusual at this point to realize that there is some functionality that every subclass must have, but that every subclass should do in its own unique way.

For example, you might be writing classes to draw charts. (We won't cover graphics programming until [Chapter 14, "Painting."](#) For now, the important point is the structure, not the content, of the code.) You might decide to create a `Chart` superclass with subclasses, as shown in [Figure 9.7](#).

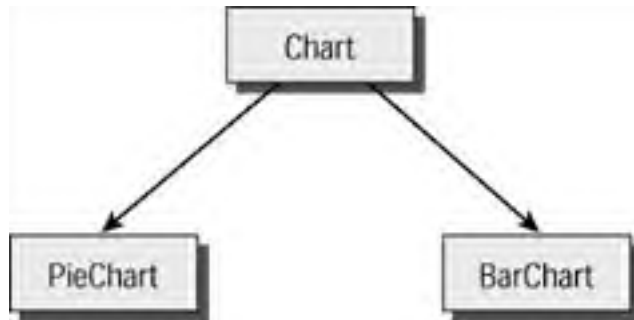


Figure 9.7: Chart class and subclasses

The class will have an array of floats, called `values`, whose values are the values to be charted. The class will also need an array of colors, called `colors`, since the color scheme should be flexible according to the user's taste. Java actually provides you with a class called `Color`. Again, you won't see this class in detail until [Chapter 14](#). For now, you only need to know that the class exists. Its package is `java.awt`, so the code examples that follow all import `java.awt.Color`.

The most important superclass method will be `display()`, whose argument is an array of float values to be charted. An auxiliary method will be `setColorScheme()`, whose argument is an array of colors. Another auxiliary method will be `useColor()`, which has an `int` argument. The argument is an index into the color scheme array. Anything subsequently drawn on the screen, until the next call to `useColor()`, will appear in the specified color.

So far the superclass looks like this:

```
package graphics;
import java.awt.Color;

public class Chart
{
    private float[] values; // Chart these values
    private Color[] colors;

    public void setValues(float[] vals)
    {
        values = vals;
    }

    public void setColorScheme(Color[] newColors)
    {
        colors = newColors;
        display(values);
    }

    private void useColor(int colorIndex)
    {
        // Never mind how this works.
        // You'll see in chapter 14.
    }

    public void display(float[] values)
    {
        for (int i=0; i<values.length; i++)
        {
            useColor(i);
            // ??? Now what ??
        }
    }
}
```

The `useColor()` method is private, since it is for use inside this class only. The other methods are public, since any user might want to call them. The problem is the "???" line in `display()`. It's obvious that for each value to be charted, you should set the appropriate color and then draw a region. You know how to set the color. Ignoring for the moment that you won't learn how to draw on the screen until later in the book, we have a deeper problem. A bar chart and a pie chart draw value regions in different ways. The object-oriented approach tells us that the individual subclasses should encapsulate the knowledge of how to draw appropriately.

Let's convert `Chart` to an abstract class:

```
package graphics;

public abstract class Chart
{
    private float[] values; // Chart these values
    private Color[] colors;

    public void setValues(float[] vals)
    {
        values = vals;
    }

    public void setColorScheme(Color[] newColors)
    {
        colors = newColors;
        display(values);
    }

    private void useColor(int colorIndex)
    {
        // Never mind how this works.
        // You'll see in chapter 15.
    }

    public void display(float[] values)
    {
        for (int i=0; i<values.length; i++)
        {
            useColor(i);
            paintRegion(i, values[i]);
        }
    }

    protected abstract void paintRegion(int n, float value);
}
```

You have added an abstract method: `paintRegion()`. Since the class now contains an abstract method, the class itself must be abstract. Any subclass that doesn't want to be abstract will have to provide an implementation of `paintRegion()`. Since only a non-abstract class can be constructed, the `display()` method in this superclass can trust that it can safely call `paintRegion()`. The true class of the executing object will never be `Chart`. It will be `BarChart`, or `PieChart`, or perhaps some other class to be written in the future. (In the last case, the new class might not be in the same package as the superclass. That's why `paintRegion()` is protected.)

The non-abstract subclasses won't look very interesting, because all their functionality is graphical. Graphical code won't make any sense to you for another few chapters, so here you're just going to see the skeletons of the classes. Here is `PieChart`:

```
package graphics;

public class PieChart extends Chart
{
    protected void paintRegion(int n, float value)
    {
        // Details not shown. Paint a pie wedge.
    }
}
```

And here is `BarChart`:

```
package graphics;

public class BarChart extends Chart
{
    protected void paintRegion(int n, float value)
    {
        // Details not shown. Paint a bar.
    }
}
```

Abstract superclasses provide an elegant structure for partitioning shared and unique functionality.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Suppose package `superpack` contains subpackage `subpack`. Suppose a source file contains the following line:

```
import superpack.*;
```


Will this line import classes in `subpack`? Write code to support your answer.
2. Create a class that illegally tries to read a private variable of another class. What is the point of this exercise?
3. Create a class that illegally tries to call a default-access method of another class.
4. Create a class that illegally tries to write a protected variable of another class.
5. True or false: If a class has at least one abstract method, the class must be abstract. Write code to support your answer.
6. True or false: If a class is abstract, it must have at least one abstract method. Write code to support your answer.
7. Write an application that tries to construct an instance of an abstract class. Can you compile the application? Can you execute it?

Chapter 10: Interfaces

The [previous chapter](#) showed you how an abstract class is a class where something is missing. This chapter will present interfaces. An interface is not actually a class, but it's like a class where nearly everything is missing.

A List of Method Declarations

An interface is mostly a list of public method declarations. The source code for an interface is similar to the source for a class in several ways. In particular, an interface definition goes in its own source file, and the source file name should be the interface name, plus `.java` at the end. When the code is compiled, the output file name is the interface name plus `.class`.

Here is an example of an interface. It should appear in source file `Talker.java`, and compilation will produce `Talker.class`:

```
package nature;
public interface Talker
{
    void say(String sayThis);
    void repeat(String repeatThis, int nTimes);
}
```

This interface is in a package called `nature`. Like a class, an interface can belong to a package. It is designated public, so it can be used by any code anywhere. If it were not public, it could be used with the `nature` package only. Interfaces cannot be private or protected.

Before we proceed, it's time to say a little about the `String` class, which appears in the argument list of both methods (and as an array in the argument list of every `main()` method). This is one of many useful utility classes that come with Java. You'll learn about them in [Chapter 12, "The Core Java Packages and Classes"](#). For now, be aware that an instance of the `String` class encapsulates a "run" or "string" of text. The data and methods of `String` won't be used in this chapter.

The list of method declarations appears between the curly brackets that follow the interface name. These declarations are much like abstract method declarations. The return types, method names, and argument lists are present, but the method body is absent, replaced by a semicolon. Unlike abstract methods, the methods declared in an interface are all public. You can declare them as public explicitly if you like, but you may not declare them private or protected. Omitting the access modifier results in public, rather than default, access.

Any class can declare that it implements any interface. This declaration occurs in the class's definition file. The class name is followed by the keyword `implements`, followed by the interface name. For example:

```
package nature;
class Parrot extends Bird implements Talker
{
    . . .
}
```

When a class declares that it implements an interface, the class is saying that it contains an implementation for each of the methods in the interface. (If this is not the case, the class will not compile.) So if the `Parrot` class compiles, you know that it contains a method called `say()` and another method called `repeat()`, with argument lists as specified in the interface.

A class is allowed to implement multiple interfaces. To do this, just provide a comma-separated list of interfaces after the `implements` keyword. So if `Flyer` and `BugEater` are interfaces of the `nature` package, you could have the following class:

```
package nature;
class Mynah extends Bird
    implements Talker, Flyer, BugEater
{
    . . .
}
```

This class would have to provide implementations for the methods of all three interfaces.

Using Interfaces

To see why interfaces are useful, consider the class inheritance hierarchy shown in [Figure 10.1](#).

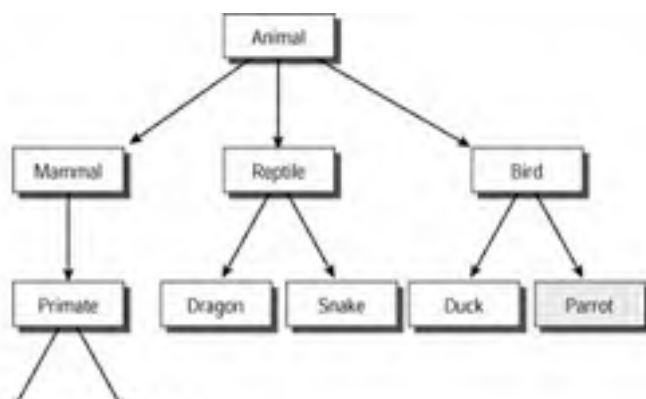




Figure 10.1: Animal kingdom class inheritance

Figure 10.1 would be a very natural way to organize your work if you were creating an extensive library of classes to model the behavior of different kinds of animals. You can imagine behavior such as live birth, cold-bloodedness, and flight being implemented by methods in the `Mammal`, `Reptile`, and `Bird` classes, respectively, and inherited by their respective subclasses. However, there is some behavior that does not fit into the inheritance model.

The three shaded classes in Figure 10.1 share somewhat related behavior. All three species are capable of speech, although the nature of that speech varies greatly from species to species. We humans speak out loud, and we understand what others say. Gorillas can't speak out loud because they don't have the right kind of vocal cords, but they can be taught to use and respond to sign language. (See www.gorilla.org for more information.) Parrots can speak out loud (and do so long after the charm has worn thin), but they do it without comprehension.

A good computer model of the animal kingdom should include speech, so it would make sense to give each of the `Human`, `Gorilla`, and `Parrot` classes its own version of the `say()` and `repeat()` methods described in the previous section. But in that case, the three classes can declare that they implement the interface. For example, `Gorilla` might look like this:

```
package nature;
public class Gorilla extends Primate implements Talker
{
    public void say(String sayThis)
    {
        // Complicated code to produce sign language.
    }

    public void repeat(String repeatThis, int nTimes)
    {
        for (int i=0; i<nTimes; i++)
            say(repeatThis);
    }
    . . .
}
```

So far we have not mentioned any benefit associated with declaring that a class implements an interface. To understand the benefit, let's revisit the issue of objects and references.

Objects and References

You have already seen that a reference is something that uniquely identifies an object. In Java, you don't have variables that are objects. Instead, you have variables that are *references* to objects. In Chapter 8, "Inheritance," you saw that the type of a reference can be different from the class of the object it refers to. This point is important enough that we make a distinction between the *type* of a reference and the *class* of an object. A reference's type is what appears in the declaration of the reference variable; an object's class is the class of the constructor that was invoked when the object was created.

You have already seen that when a reference points to an object, the type of the reference can be exactly the class of the object, or it can be any superclass of the class of the object. Interfaces provide an even wider range of reference types, because reference variable types can be interfaces as well as classes. An interface-type reference can legally point to an object if that object's class implements the interface. So in our ongoing example, the following code would be perfectly legal:

```
Parrot polly = new Parrot();
Talker aTalker = polly;
```

Or even:

```
Talker aTalker = new Parrot();
```

With an interface-type reference, you can call only the methods of the interface. This may seem limiting, but the benefit is huge. Consider the following method:

```
singHappyBirthday(Talker t, String forWhom)
{
    t.repeat("Happy birthday to you.", 2);
    t.say("Happy birthday, dear");
    t.say(forWhom);
    t.say("Happy birthday to you.");
}
```

This method has a talker recite the song, no matter what the class of the talker. A human will sing, a gorilla will sign, a parrot will squawk. This is another example of polymorphism: A single method name (`say`, and also `repeat`) appears in many different forms.

instanceof

Java has a keyword, `instanceof`, that tests the relationship between an object and a reference type. The syntax is

```
<reference> instanceof <type>
```

The reference can be any reference. The type can be the name of any class or interface. The value of an `instanceof` expression is boolean. It is true if a reference of the given type legally can point to the object pointed to by the given reference. For example, this code will print out the message:

```
Duck daffy = new Duck();
if (daffy instanceof Bird)
{
    System.out.println("Yes, Daffy is a bird.");
}
```

So will the following:

```
Bird daffy = new Duck();
if (daffy instanceof Bird)
{
    System.out.println("Yes, Daffy is a bird.");
}
```

And so will the following:

```
Object daffy = new Duck();
if (daffy instanceof Bird)
{
    System.out.println("Yes, Daffy is a bird.");
}
```

The `instanceof` keyword doesn't care about the type of the variable (that is, the word that comes before `instanceof`). What matters is the class of the object to which the variable points, and in all three examples the class is `Duck`. When the second argument of `instanceof` is a class, as in this example, the value is true if the object's class is the same as, or a subclass of, the second argument. Here is an example of `instanceof` where the second argument is an interface:

```
Duck daffy = new Duck();
if (daffy instanceof Talker)
{
    System.out.println("Yes, Daffy is a bird.");
}
```

When the second argument is an interface, the value of an `instanceof` expression is true if the object's class implements the interface. Here, the object's class is `Duck`, which does not implement `Talker`, so the value is false. In the following code, the value is true:

```
Gorilla ndume = new Gorilla();
if (ndume instanceof Talker)
{
    System.out.println("Yes, Ndume can talk.");
}
```

Data in Interfaces

An interface is allowed to contain data, provided the data is public, final, and static. This provides an easy way to define constant data.

In [Chapter 9, "Packages and Access,"](#) you looked at static data within a class with the following example:

```
class Zebra extends Mammal
{
    static private final double KGS_TO_LBS = 2.2;
    . . .
}
```

The variable `KGS_TO_LBS` can be used anywhere within the `Zebra` class. If other classes in the same package want to use the constant, you can declare `KGS_TO_LBS` to have default access (or even protected access, in which case the other-package subclasses can also use it). The other classes can refer to the constant as `Zebra.KGS_TO_LBS`. Sometimes this is fine, but in our example it seems to imply that converting from kilograms to pounds has something to do with zebras. If a constant is more properly associated with a package in general, rather than with any individual class, generally it is better to put it in an interface.

For example, you might be creating a package of classes that model the physics of various mutually interacting heavenly bodies. Your package would be called `astro`, and the classes would be `Planet`, `Star`, `BlackHole`, `Comet`, and so on. (Their official names would be `astro.Planet`, `astro.Star`, `astro.BlackHole`, and `astro.Comet`.) The classes probably would all need to use certain fundamental constants, such as the speed of light and the mass of a proton. Let's look at the various options for implementing these.

First, you can avoid the use of constants altogether. Wherever you need the speed of light, use `3.0e8`; wherever you need the mass of a proton, use `1.67e-27`. As you saw in [Chapter 9](#), this approach is risky. If you type a wrong digit, you'll introduce a bug that can be very hard to find. Moreover, readers of your code might not recognize the significance of `3.0e8` or `1.67e-27` (would you?), so they would not understand the formulas you were implementing.

The next step is to put constants in one of your classes. You might pick `Star`, arbitrarily, and insert the following lines:

```
final static double LIGHT_SPEED = 3.0e8;
final static double PROTON_MASS = 1.67e-27;
```

By convention, constants are in all capital letters, with words separated by underscores. Recall that with constants, a typing error results in a variable name that the compiler will not recognize, so you recruit the compiler to help you find typos.

Before we go further, notice that the constant names can be improved on. As they stand, they are truthful but not entirely helpful. `1.66e-27 whats?` `3.0e8 whats per what?` For optimum clarity, it's best to put the units in the constant names:

```
final static double LIGHT_SPEED_M_PER_SEC = 3.0e8;
final static double PROTON_MASS_KG = 1.67e-27;
```

That takes a little more typing, but now nobody will ever think the speed of light is expressed in miles per second, or proton mass in micrograms. Within the `Star` class, you can refer to the constants by name. Elsewhere in the `astro` package, you can refer to them as `Star.LIGHT_SPEED_M_PER_SEC` and `Star.PROTON_MASS_KG`.

This is certainly better than typing literal constants, but it implies that the constants are somehow naturally associated with the class they appear in. You can go one step further by creating an interface for your constants:

```
package astro;

interface AstroConstants
{
    final static double LIGHT_SPEED_M_PER_SEC = 3.0e8;
    final static double PROTON_MASS_KG = 1.67e-27;
}
```

You can also put method declarations in the interface code, but you don't have to. An interface can declare any number of methods, including zero. Now classes in the `astro` package can refer to `AstroConstants.LIGHT_SPEED_M_PER_SEC` and `AstroConstants.PROTON_MASS_KG`. You don't have to pick a class arbitrarily to put your universal constants in.

You can go one step further, because of the following rule: A class that implements an interface can use the constants of that interface by name, without prefixing the interface name. So your `BlackHole` class could use the following declaration:

```
package astro;

class BlackHole implements AstroConstants
{
    . . .
}
```

You don't have to do any work to ensure that `BlackHole` implements all the methods of the interface, because there are no methods in the interface! And now, anywhere within the `BlackHole` code, and within the code of any other class that implements `AstroConstants`, you can refer simply to `LIGHT_SPEED_M_PER_SEC` and `PROTON_MASS_KG`.

Warning Beware of a subtlety concerning interfaces. All methods and constants in an interface are public. You can use the `public` keyword explicitly for clarity, but if you omit it, the interface's features are still public. They do not have default access, which is what you get if you omit an access modifier in the source code for a class. In an earlier example, when you put the constants in class `Planet`, you didn't use an access modifier. So the constants had default access and could be used anywhere within the `astro` package, but nowhere else. When you moved the constants to the interface, they were forced to be public. Hence, they were accessible from any code regardless of package.

Extending Interfaces

An interface is allowed to extend another interface. The syntax is

```
interface <interface_name> extends <parent_interface>
{
    // Declarations and data
}
```

The new interface consists of all the methods and data defined in the parent interface. For example, you might define the following:

```
package nature;

interface SingingTalker extends Talker
{
    public void sing(String song);
}
```

This interface consists of the two methods defined in `Talker` (`say()` and `repeat()`), as well as `sing()`. A class that wants to implement this interface must use all three methods.

A class can extend only a single parent class, but an interface can extend multiple parent interfaces. For example, if `InterA`, `InterB`, and `InterC` are all interfaces, the following is legal:

```
interface ManyParents extends InterA, InterB, InterC
{
    public int anotherMethod(double d, char ch);
}
```

This interface consists of all the constants and methods of all three parents, plus the method defined explicitly in the source code.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Suppose an interface declares three methods. And suppose a class declares that it implements the interface, but in fact it only implements two out of the three methods. What happens when you try to compile the class? (The way to answer this question, of course, is to write an interface and a class.)
2. If class A implements an interface, any subclasses of A inherit all the methods specified in the interface. Does this mean that subclasses of A also implement the interface? Write code to discover the answer.

3. Given the following interface:

```
interface InterfaceQ3
{
    void printALine();
}
```

Will the following code compile?

```
class ClassQ3 implements InterfaceQ3
{
    void printALine()
    {
        System.out.println("OK");
    }
}
```

4. Don't worry, the following question requires absolutely no understanding of physics. In fact, it might make you grateful that you chose computer programming instead. Suppose you have the following interface:

```
package physics;
interface PhysicsConstants
{
    public static final double ELECTRON_MASS_KG = 9.11e-31;
    public static final double
        STEFAN_BOLTZMANN_CONSTANT_WATTS_PER_M2 = 5.67e-8;
}
```

What does the following application print out?

```
package physics;

public class Q4 implements PhysicsConstants
{
    public static void main(String[] args)
    {
        System.out.println("The value is " +
            STEFAN_BOLTZMAN_CONSTANT_WATTS_PER_M2);
    }
}
```

Chapter 11: Exceptions

Overview

At several points in this book, you have seen how certain program features alter the usual linear flow of program execution. You saw that loops are like whirlpools, conditional statements are like forks in the road, and method calls are like detours.

Extending this analogy, exceptions are like jumping through hyperspace. With exceptions, you can instantly end up far from where you began, with no prospect of getting back. Of course, jumping through hyperspace is an unusual way to travel. And as you might expect, exceptions are to be used only in unusual programming circumstances.

This chapter will show you how to use exceptions to indicate unusual conditions in the state of your programs. It will take a careful approach to this topic, because exceptions are complicated. It would not be helpful to overwhelm you with information. First you'll look at some of the basic concepts of exceptions. Then you'll learn how exceptions are used in real life.

Exceptions Oversimplified

This chapter will begin with an explanation of why exceptions are important. Then it will look at an extended example that uses exceptions. Please bear in mind that the example code here is intended to demonstrate concepts, not to stand as an example of good programming. Later on, once you understand how exceptions affect program flow, you'll see more realistic code examples.

The Trouble with Error Codes

First, let's look at why Java needs a feature to support unusual program status. Imagine a remote database that stores daily rainfall reports for various weather stations. (A remote database is one where the data is stored on a different computer from the one you are using. The two machines are connected by a network.) Without going into the details of how to get data from a remote database into a Java program, imagine a class with a method that somehow retrieves rainfall numbers. The method might be called

```
float getRainfall(int station, int year,
                 int month, int day)
```

For example, to get the rainfall for station #7 for July 8, 2001, you would call

```
getRainfall(7, 2001, 7, 8);
```

You can imagine the method doing network connection stuff, database login/password stuff, database query stuff, network disconnection stuff, and finally returning a value. The problem is, what happens if something goes wrong with the database? Here are a few things that could go wrong:

- The database computer could be turned off.
- The network cables could break.
- The database could be deleted.
- The database management code could crash.
- The database password could be changed.

All of these possibilities are beyond the control of the programmer who is writing the `getRainfall()` method. They are not bugs. A bug is when you write code that doesn't do what you want it to do. Bugs can be avoided by intelligent design and programming, but no amount of programming forethought on your part can prevent someone from walking up to a remote computer and turning it off.

Your code has no straightforward way to deal with these unusual circumstances. There is no straightforward way to tell the method's caller that the database computer was turned off or the password didn't work. Before the invention of exceptions (which predate Java), the only reasonable option was to designate certain special return values, called error codes, to indicate that something unusual happened.

In this example, you might reserve large return values as error codes. You might decide that 10,000 means the password didn't work, 20,000 means the network was unresponsive, and so on. After all, if it ever rains 10,000 inches in a single day anywhere on Earth, we'll all have more pressing problems than data processing to worry about.

What is wrong with this approach? The problem is that anyone, anywhere, who calls `getRainfall()` has to remember to deal with all the error codes:

```
float rainfall = getRainfall(7, 2001, 7, 8);
if (rainfall < 1000)
{
    // Process normally.
}
else if (rainfall == 1000)
{
    // Deal with password problem.
}
else if (rainfall == 20000)
{
    // Deal with unresponsive net.
}
```

The error-processing code might display a message, or it might be more sophisticated. The password-handling code might try a different password. The network-handling code might retry the query at one-second intervals, or it might page a system administrator. But no matter how the errors are handled, anyone who calls the method has to check all return values to see if an error code was returned. To do this, they need good documentation that describes each code and its meaning. The programming language can do nothing to support error handling, because from the compiler's point of view, an error code is just an ordinary value returned by a method.

Special problems are introduced when the method is revised, if the new revision introduces new error codes. Now all the old documentation is incomplete. It's even worse if the new rev of the method changes the significance of an existing code.

Things can get worse yet. What if you realize that your error codes actually represent legitimate return values? Certainly, there's nowhere on Earth where it rains 10,000 inches in a day. But on other planets, with active atmospheres and extremely long days (Mercury, for example), 10,000 is common.

Less imaginatively, the data might be gathered automatically into the database by electronic rain gauges. If the electronics fail, a gauge could erroneously report a measurement of 10,000. Then, when `getRainfall()` returned the value, the error-handling code might page a system administrator, who would be paid overtime for rushing to the office at 4:30 in the morning. The administrator would spend hours determining that the network was healthy, and would probably be grouchy for the rest of the day.

You have probably realized that rainfall can never be less than zero, so you should have reserved negative error codes, rather than large ones. That would make your method interplanetary, but the other problems would remain. Still, unless you use exceptions, error codes are the only option. By the time you finish this chapter, you should be an enthusiastic user of exceptions.

Throwing Exceptions

In this section, you will meet two new Java keywords: `throw` and `throws`. They look almost identical, but `throw` only appears in executable code, while `throws` only appears in method declarations. You will also meet the `Exception` class. By now, you are aware that Java uses a number of classes that are provided for you by the system. The `Object` class is an obvious example, and you have also seen a little bit of the `String` class. In [Chapter 12, "The Core Java Packages and Classes,"](#) you will learn about more of these classes. They are too numerous to describe in detail, but you will also learn where to find out about provided classes as needed. But in order to learn that, you first have to understand exceptions.

To see exceptions in action, let's change the `getRainfall()` example. For now, let's suppose that only one unusual condition is recognized by the code: a crashed database. To detect this condition, assume you have a boolean method called `databaseOk()` that returns true if the database is healthy and false if it has crashed. If you were to use the error-code approach, you might write the following:

```
1. float getRainfall(int station, int year,
2.                  int month, int day)
3. {
4.     if (databaseOk() == false)
5.         return -1;
6.
7.     // Get & return rainfall from db
8. }
```

You use `-1` as an error code to indicate a crashed database. Now here is the same method, rewritten to use an exception:

```
1. float getRainfall(int station, int year,
2.                  int month, int day) throws Exception
3. {
4.     if (databaseOk() == false)
5.     {
6.         Exception x = new Exception("The db crashed.");
7.         throw x;
8.     }
9.     // Get & return rainfall from db
10. }
```

This changes the code in 3 ways:

- It adds `throws Exception` to the declaration on line 1.
- It creates an instance of the `Exception` class on line 6.
- It throws the exception (whatever that means) on line 7.

The addition of `throws Exception` to the declaration announces that this method now might throw an exception. Any particular call to the method might or might not throw, but any code that calls the method must be prepared to deal with the possibility.

Line 6 is just an ordinary constructor call. Until they are thrown, exceptions are just ordinary objects. There are two commonly used versions of the `Exception` constructor: a no-args version, and the version used here, which takes a string of text as an argument. The text can be retrieved later by calling the exception's `getMessage()` method. Later on, you will see how this is useful when processing exceptions.

Line 7 is the big idea. The `throw` keyword must be followed by an exception. (Strictly speaking, a few other things can follow `throw`, but they are beyond the scope of this book.) When the `throw` statement is executed, the current execution of the current method is abandoned immediately. Execution jumps (as if through hyperspace!) to the appropriate exception-handling code for the particular exception that was thrown. The exception handler uses the `catch` keyword, which is presented in the [next section](#).

Catching Exceptions

It stands to reason that things that are thrown ought to be caught. This is true for balls, Frisbees, kisses, and exceptions.

When you call a method that declares that it throws an exception, the calling code can't just call the method. For example, the following code will not work:

```
float rainfall = getRainfall(7, 2001, 7, 8);
```

This call used to be fine, but now `getRainfall()` throws an exception, which any calling code must be prepared to catch. The call has to look something like this:

```
1. try
2. {
3.     float rainfall = getRainfall(7, 2001, 7, 8);
4.     System.out.println("rainfall was " + rainfall);
5. }
6. catch (Exception x)
7. {
8.     System.out.println("getRainfall() failed.");
9.     System.out.println("Message is: " + x.getMessage());
10. }
11. System.out.println("And life goes on.");
```

The code on lines 2-5 (in the curly brackets immediately after `try`) is called a *try block*. There are two rules to know about try blocks:

- Any code that throws an exception must appear in a try block. (For now. Later you'll learn how to get around this rule.)
- At least one statement in the try block must throw an exception.

The code on lines 7-10 (in the curly brackets immediately after the `catch` line) is a *catch block*. When a statement in a try block is executed and causes an exception to be thrown, the current pass through the try block is abandoned immediately. Execution jumps to the first line of the catch block.

Note the code in parentheses on line 6, after `catch`. It looks like a variable declaration, and indeed it is. When an exception is thrown, the JVM makes the exception object accessible to the catch block. When you declare `Exception x`, you are saying that you want to use the variable name `x` as the name of your reference to the exception. This variable has scope (that is, valid meaning) only within the catch block.

Notice line 9, which makes a method call on `x`. Recall that you can pass a message into the `Exception` constructor. The message is stored in the exception object, and the `getMessage()` call retrieves it. Recall from the [previous section](#) that `getRainfall()` stored a message that said, "The db crashed."

Note If you are at all uncomfortable with the way line 9 adds literal text to a method call, please be patient. All will be explained in the [next chapter](#). For now, just be aware that it works.

What happens if the try block runs in its entirety, with no exception being thrown? In this case, the catch block is ignored. Execution jumps around the catch block, from line 4 to line 11. If this is the case, and if the rainfall value is 1.45 inches, the code will produce the following output:

```
Rainfall was 1.45  
And life goes on.
```

On the other hand, if the call to `getRainfall()` throws an exception, the output will be
`getRainfall()` failed.
Message is: The db crashed.

The Simple Exception Lab animated illustration demonstrates the flow of execution through code, which is almost identical to this example. To run the program, type `java exceptions.SimpleExceptionLab`. You will see the display shown in [Figure 11.1](#).

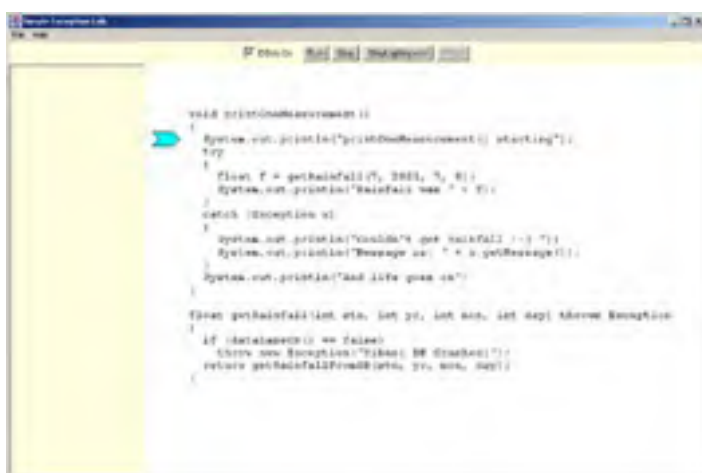


Figure 11.1: Simple Exception Lab

You will see the code for a method that calls another method to retrieve a rainfall measurement from an imaginary remote database. (There isn't really a remote database, and your computer doesn't have to be connected to a network for the animated illustration to work.) You can use the checkbox to control whether the imaginary database is working or not. As [Figure 11.1](#) shows, the DB (database) is initially okay. If you uncheck the checkbox, the second method will throw an exception that will be caught by the first method. The text area to the left of the display will show all output from the `println` statements. Try running the program once with the DB is Ok checkbox checked, to simulate normal execution. [Figure 11.2](#) shows the final state.

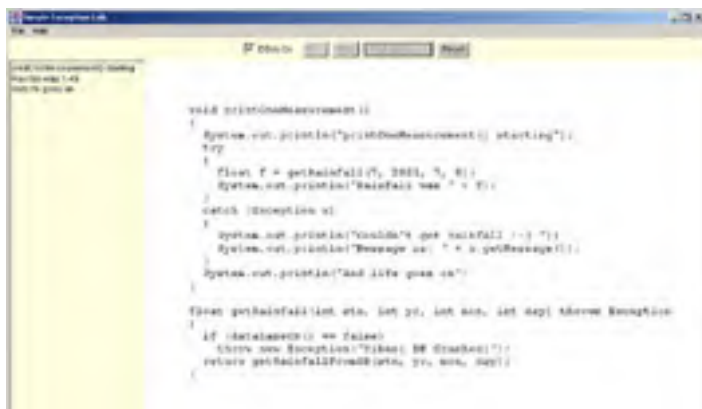




Figure 11.2: Simple Exception Lab: final state with normal execution

Think about what the code would print out if the database wasn't okay. Run it again with the checkbox unchecked to observe the error-handling behavior. Was the output what you expected?

Team LIB

PREVIOUS NEXT

Exceptions in the Real World

So far, the point of this chapter has been to familiarize you with exceptions, and especially with trying, throwing, and catching. Now that you understand these concepts, it is time to tell you that the situation is actually a lot more complicated. There are many kinds of exceptions, each one indicating a different kind of problem, and each one capable of being handled separately.

The remainder of this chapter will show you how to deal with the multitude of real-world exceptions.

Two Families of Exceptions

The `Exception` class has more than 100 subclasses in the core Java packages. (The *core packages* are the ones that you get along with the JVM and the Java compiler. You can think of them as the infrastructure of Java, providing classes that are essential to the operation of the JVM, the compiler, and your own applications.)

The two families are

- Checked exceptions
- Runtime exceptions

The `Exception` class has a subclass called `RuntimeException`. The family of runtime exceptions consists of the `RuntimeException` class and all its subclasses. The family of checked exceptions consists of all other exception classes, including `Exception` itself. (Note that there is no `CheckedException` class.)

Generally, exception classes have long and descriptive names, such as `PrinterIOException` and `ArrayIndexOutOfBoundsException`. Usually, the class name tells you very specifically what went wrong. Let's use these two classes to look at the difference between checked and runtime exceptions.

`PrinterIOException` is a checked exception. It's thrown by methods that interact with a printer. If a printer is jammed, unavailable, or in some other failure state, the method throws `PrinterIOException`. `ArrayIndexOutOfBoundsException` is a runtime exception. As you can guess from the name, it is thrown when an array index is \geq the length of the array, or when the index is negative.

What's the difference between these two situations? It all comes down to who is responsible for creating the problem. In the case of `PrinterIOException`, you can't really say it's anyone's fault. Printers jam up or fail in other ways that are familiar to all owners of printers. That's an environmental hazard. It's unavoidable, like bad weather. On the other hand, with `ArrayIndexOutOfBoundsException`, it's easy to assign blame. The programmer who wrote the line of code that used the illegitimate array index should have done a better job. After all, it would be ridiculous to tell you to turn to [page 1,963](#) in this book... or worse yet, to page -47. Similarly, you shouldn't refer to an array element that doesn't exist.

To generalize from these examples: All checked exceptions represent situations that are unavoidable. All runtime exceptions represent situations that can be avoided by better programming. This implies that your Java programs might sometimes throw checked exceptions, but they should never throw runtime exceptions.

The proper way to deal with checked exceptions is with the try/catch mechanism described earlier in this chapter. The proper way to deal with runtime exceptions is... well, you should never have to deal with them, because your code should never throw them. Of course, code is never perfect the first time you write it. Whenever you write a long piece of code, your first job is getting it to compile. Once you do that, you're only halfway finished. The next step is to make your code run correctly by finding and eliminating bugs. During this phase of development, you are likely to encounter runtime exceptions, and your job is to eliminate them. So your finished, polished, ready-for-market code should never throw runtime exceptions. During development, runtime exceptions are signposts that point to code that needs fixing.

Runtime exceptions should *not* be caught in catch blocks. But how can this be? Earlier in this chapter, you learned that if code might throw an exception, the code has to appear in a try block and the exception has to be caught in a corresponding catch block. Well, that was an oversimplification to avoid giving you too much information all at once. Now that you're half an expert on exceptions, you can learn the whole story.

Code that throws checked exceptions *must* appear in a try block, with the exception caught in a catch block. But this rule *does not apply* to code that throws runtime exceptions. Such code *may* appear in a try/catch structure, but it doesn't have to, and usually it should not. Instead, the code that would throw the runtime exception should be fixed so that it no longer throws.

Runtime Exceptions and Stack Traces

Now you know that you should *not* catch runtime exceptions. But then what happens when one is thrown?

When any kind of exception is thrown, the JVM stores some very useful information in the exception. This information is called the *stack trace*, and often it's all you need to find the source of the problem. The stack trace tells you what line of code threw the exception, as well as the name of the method that contains the line. The stack trace also tells you what line of code called that method, and so on. It goes back and back until you get the line in your `main()` method that called the method that called the method that called the method that owned the line that threw the exception. It's like *This Is the House That Jack Built*, only it's about a Java program instead of a house:

This is the program that you built.

This is the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

This is the line that threw the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

This is the method that owns the line that threw the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

This is the line that calls the method that owns the line that threw the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

This is the method that owns the line that calls the method that owns the line that threw the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

...

This is the `main()` method that owns the line that calls the method that owns the line that calls the method that owns the line... that threw the `ArrayIndexOutOfBoundsException` that was thrown from the program that you built.

Let's look at a practical example. Suppose you have the following application:

```
1. public class ShowMeATrace
2. {
3.     public static void main(String[] args)
4.     {
5.         int[] cubes = new int[10];
6.         storeCubes(cubes);
7.     }
8.
9.     private static void storeCubes(int[] intArr)
10.    {
11.        for (int i=0; i<=10; i++)
12.            storeOneCube(intArr, i);
13.    }
14.
15.    private static void storeOneCube(int[] ints,
16.                                     int index)
17.    {
18.        ints[index] = index*index*index;
19.    }
20. }
```

The `main()` method creates an array that's passed to `storeCubes()`. The `storeCubes()` method loads each array component with the cube of its index. It does this by calling `storeOneCube()` once for each component. When you run this application, you get the following output:

```
java.lang.ArrayIndexOutOfBoundsException
    at ShowMeATrace.storeOneCube(ShowMeATrace:18)
    at ShowMeATrace.storeCubes(ShowMeATrace:12)
    at ShowMeATrace.main(ShowMeATrace:6)
Exception in thread "main"
```

This output is a stack trace. Reading from top to bottom, you find that an `ArrayIndexOutOfBoundsException` was thrown from line 18 in the `storeOneCube()` method. The offending call to `storeOneCube()` was made on line 12 in `storeCubes()`, which was called from line 6 in `main()`. (By the way, notice that the first line of the trace implies that `ArrayIndexOutOfBoundsException` belongs to the `java.lang` package. The core Java classes belong to a package called `java`, which contains many subpackages. The most important subpackage is `java.lang`, which contains a large number of vital infrastructure classes. You will look at some of these classes in the [next chapter](#).)

So the stack trace tells you to pay attention to lines 18, 12, and 6. Usually your best strategy is to look at lines in the order they appear in the trace. Line 18 seems innocent, as long as `index` is reasonable. But `index` is supplied by the method's caller, so you look at line 12. You see that `index` in `storeOneCube()` corresponds to `i` in `storeCubes()`. The maximum value of `i` is 10, but the array only has 10 components, so the maximum legal index is 9. You have found the problem.

There are two ways to fix the bug. The lazy way would be to change line 11 like this:

```
for (int i=0; i<10; i++)
```

That would solve the problem at hand, but if the array size (in `main()`) ever changes, you will have to remember to change line 11. The safe way, which is better style in all cases, is to use the following for line 11:

```
for (int i=0; i<intArr.length; i++)
```

If you have a program that uses an array, it is very likely that eventually you will create a for loop to do some kind of processing on each array component. If you use the kind of for loop shown here, you will always be sure to process every component while avoiding `ArrayIndexOutOfBoundsException`.

Warning Be aware that some versions of the JVM do not provide stack traces when exceptions are thrown. This usually happens because the JVM performs some kind of optimization that makes it impossible to piece together the stack trace information. When these machines throw an exception, you just get a message that tells you the class of the exception.

Checked Exceptions

In the [previous section](#), you saw that you should not catch runtime exceptions, even though the language allows you to. However, when you call a method that throws a checked exception, you have no choice but to use the try/catch mechanism. If you don't, your code will not compile.

Suppose you have a method, called `printRetAddr()`, that prints your return address on an envelope. Assume you have the kind of printer that can detect whether it is loaded with paper or envelopes. If it is not loaded with envelopes, the method throws `PrinterIOException`, which is a checked exception. If you want to call the method, your code might do the following:

```
void printSomeEnvelopes(int nEnvelopes)
{
    for (int i=0; i<nEnvelopes; i++)
        printRetAddr();
}
```

Simple enough, but it won't compile. Your compiler error will be something like this:

PrinterIOException must be caught or declared to be thrown at line xx, column xxx.

This tells you that you have two options. Your first option is to put the call to `printRetAddr()` inside a try block:

```
void printSomeEnvelopes(int nEnvelopes)
{
    for (int i=0; i<nEnvelopes; i++)
    {
        try
        {
            printRetAddr();
        }
        catch (PrinterIOException piox)
        {
            System.out.println("Please load printer " +
                               "with envelopes.");
        }
    }
}
```

Earlier in this chapter, you saw code that catches `Exception`. Here you see that any subclass of `Exception` may be caught (as long as it really is thrown in the try block; see Exercise 4 at the end of this chapter). The catch block will be executed if the try block causes a `PrinterIOException` to be thrown.

You have a second option. If you don't want to use try/catch, you can simply declare that `printSomeEnvelopes()` throws `PrinterIOException`:

```
void printSomeEnvelopes(int nEnvelopes)
    throws PrinterIOException
{
    for (int i=0; i<nEnvelopes; i++)
        printRetAddr();
}
```

Now any method that calls `printSomeEnvelopes()` must either use try/catch or declare that it too throws `PrinterIOException`.

Multiple Catch Blocks

Typically, code in a try block can throw more than one kind of exception. To illustrate this, let's look at another type of checked exception: `ConnectException`. This is usually thrown by code that attempts to connect to a machine on the network, such as a Web server. If the remote machine does not respond (because it has been turned off, or is undergoing maintenance, or has burned up), the code that detects the lack of response should throw a `ConnectException`. (You will look at network connections in detail in [Chapter 13, "File Input and Output."](#) For now, the point is that now you know about two checked exception types.)

To extend this example, let's make `printSomeEnvelopes()` more responsible. Suppose you have two utility methods at your disposal:

`getNumEnvelopesInStock()` Returns the number of envelopes left, not counting the ones you just printed. This value is retrieved from a remote database.

`setNumEnvelopesInStock()` Updates the number of envelopes left. This value is stored on the remote database.

Both methods throw `ConnectException` if the machine where the remote database resides cannot be contacted. Now `printSomeEnvelopes()` can be written like this:

```
void printSomeEnvelopes(int nEnvelopes)
{
    for (int i=0; i<nEnvelopes; i++)
    {
        try
        {
            printRetAddr();
        }
        catch (PrinterIOException piox)
        {
            System.out.println("Please load printer " +
                               "with envelopes.");
            return;
        }
    }

    try
    {
        int nEnvelopesLeft = getNumEnvelopesInStock();
        nEnvelopesLeft -= nEnvelopes;
        setNumEnvelopesInStock(nEnvelopesLeft);
    }
    catch (ConnectException conx)
    {

```

```
        System.out.println("Couldn't connect.");
    }
}

System.out.println("printSomeEnvelopes() done.");
}
```

The second try block updates the remote database, taking into account the number of envelopes that were just printed. When `printSomeEnvelopes()` is called, there are four possibilities:

The code could run normally, with no exceptions being thrown. Neither catch block is executed. The method prints the "done" message and then returns.

A `PrinterIOException` is thrown from `printRetAddr()`. Execution jumps to the first catch block, which prints the "Please load..." message and then returns. (It returns because no envelopes were used, so the number in the database shouldn't be decremented. If the catch block did not return, the second try block would be executed.)

A `ConnectException` is thrown from `getNumEnvelopesInStock()`. Execution jumps to the second catch block, which prints the "Couldn't connect" message. Then execution continues after the catch block. The "done" message is printed, and then the method returns.

A `ConnectException` is thrown from `setNumEnvelopesInStock()`. Just as in the previous case, execution jumps to the second catch block, which prints the "Couldn't connect" message. Then the "done" message is printed and the method returns.

This code can be simplified. A single try block is allowed to throw multiple exception types, provided there is a catch block for each type. This might require multiple catch blocks for the try block:

```
void printSomeEnvelopes(int nEnvelopes)
{
    try
    {
        for (int i=0; i<nEnvelopes; i++)
            printRetAddr();
        int nEnvelopesLeft = getNumEnvelopesInStock();
        nEnvelopesLeft -= nEnvelopes;
        setNumEnvelopesInStock(nEnvelopesLeft);
    }
    catch (PrinterIOException pioex)
    {
        System.out.println("Please load printer " +
            "with envelopes.");
    }
    catch (ConnectException cx)
    {
        System.out.println("Couldn't connect.");
    }
    System.out.println("printSomeEnvelopes() done.");
}
```

The work has been consolidated into the single try block. There are two catch blocks. If the try block threw five or 50 exception types, there could be five or 50 catch blocks.

When the JVM detects a thrown exception in the try block, it scans the various catch blocks. The current pass through the try block is abandoned, and execution continues in the first catch block that is appropriate to the type of thrown exception. This version of the method behaves exactly like the previous version, but it's easier to read because all the normal execution code appears in the try block, while problems are handled in the various catch blocks. No matter how many catch blocks there are, a single thrown exception is only handled by one catch block. After the catch block runs (and it doesn't contain a `return` statement), execution continues after the last catch block.

Catch Blocks and *instanceof*

The [previous section](#) introduced multiple catch blocks. You learned that execution continues in the first catch block that is appropriate to the type of thrown exception. But what makes a catch block appropriate? You might think that the type declared in parentheses after `catch` must match the class of the exception that was thrown. But this is not the whole story. The whole story involves `instanceof`.

Recall from [Chapter 10, "Interfaces"](#), that the syntax for `instanceof` is

```
<reference> instanceof <type>
```

If the type is a class name, and the reference points to an object whose class is either the type or a subclass of the type, `instanceof` evaluates to `true`.

When the JVM looks for a catch block to handle an exception, it uses `instanceof` to determine whether or not a particular catch block is appropriate. To illustrate, let's blur the `printSomeEnvelopes()` example:

```
void printSomeEnvelopes(int nEnvelopes)
{
    try
    {
        // STUFF
    }
    catch (PrinterIOException pioex)
    {
        // STUFF
    }
    catch (ConnectException cx)
```



```
    {
        // STUFF
    }
    System.out.println("printSomeEnvelopes() succeeded.");
}
```

If the try block throws, the JVM asks if the exception is an instance of `PrinterIOException`. If so, the first catch block is executed. Otherwise, the next catch block is tested. The JVM asks if the exception is an instance of `ConnectException`. If so, the second catch block is executed. In either case, only the one catch block is executed; the other is ignored. If the executing catch block does not return, execution then proceeds at the first statement following the last catch block.

If no exception is thrown, the try block runs to completion and both catch blocks are skipped.

Sometimes you can take advantage of how the JVM determines the appropriate catch block. In the last revision of our example, the two different kinds of exceptions were handled differently, but this might not always be the case. Suppose you decide that no matter what kind of trouble crops up, `printSomeEnvelopes()` should just print a message that says "Could not print" and then return. If there is no trouble, the method should print "Succeeded."

Both `PrinterIOException` and `ConnectException` are subclasses of a common superclass called `IOException`. So `printSomeEnvelopes()` can be rewritten like this:

```
void printSomeEnvelopes(int nEnvelopes)
{
    try
    {
        for (int i=0; i<nEnvelopes; i++)
            printRetAddr();
        int nEnvelopesLeft = getNumEnvelopesInStock();
        nEnvelopesLeft -= nEnvelopes;
        setNumEnvelopesInStock(nEnvelopesLeft);
        System.out.println("Succeeded");
    }
    catch (IOException iox)
    {
        System.out.println("Could not print");
    }
}
```

Now, any kind of exception that the try block might throw will pass the instance of `IOException` test, so execution will end up in the single catch block.

You can get even more sophisticated. There is a kind of catch block that is informally called a *safety net catch block*. This is not official Java terminology, but it's very commonly used. You might have a try block that throws many subclasses of `IOException`, including `PrinterIOException` and `ConnectException`. Suppose those two types require individual handling, but all other types can be handled the same. You could do the following:

```
try
{
    // Lots
    // and
    // lots
    // and
    // lots
    // of code that throws
    // lots
    // and
    // lots
    // and
    // lots
    // of subclasses of IOException
}
catch (PrinterIOException piox)
{
    // Special PrinterIOException handling
}
catch (ConnectException cx)
{
    // Special ConnectException handling
}
catch (IOException iox)
{
    // General IOException handling
}
```

If either `PrinterIOException` or `ConnectException` is thrown, the appropriate specific catch block will be executed. If a different type of `IOException` is thrown, the JVM will first check if the exception is an instance of `PrinterIOException`. It isn't, so next the JVM will check if it is an instance of `ConnectException`. Again, it isn't, so the JVM checks if it is an instance of `IOException`. And it is, because subclasses pass the instance of test, so the last catch block is executed. You can see how the last catch block is a kind of safety net, catching all `IOExceptions` that aren't caught by the two specific catch blocks.

The safety net block could have caught `Exception` instead of `IOException`. The code would have identical behavior, but the safety net is overly general and is considered bad coding style. The exception type caught by a safety net should be the lowest-level subclass that gets the job done. See Exercise 5 at the end of this chapter to find out why.

When you use a safety net, be careful about the order of appearance of your catch blocks. Don't do the following:

```
try
{
    // Something
}
catch (IOException iox)
{
    // General IOException handling
}
catch (PrinterIOException piox)
{
    // Special PrinterIOException handling
}
catch (ConnectException cx)
{
    // Special ConnectException handling
}
```

The second and third catch blocks can never be executed, because both `PrinterIOException` and `ConnectException` pass the instanceof `IOException` test. The compiler will not allow this code. You will get a compiler message that says something like, "Catch is unreachable at line xxx."

The Advanced Exception Lab animated illustration shows exception handling in situations where the try block can throw multiple exception types from any of several lines. To start the program, type `java exceptions.AdvancedExceptionLab`. You will see the display shown in [Figure 11.3](#).



Figure 11.3: Advanced Exception Lab

You get to choose the type of exception that will be thrown. Click on the Choose Type... button and you will see a dialog that lets you choose from four checked types, as shown in [Figure 11.4](#).



Figure 11.4: Choosing an exception type in Advanced Exception Lab

You can click on any of the exception types except the `Exception` superclass. You can also choose which line throws the exception by clicking on the checkbox on the line of your choice on the main screen. The lab lets you choose one of five code configurations (via the File?Configurations menu). In each configuration, a method called `top()` calls a method called `middle()`, which calls a method called `bottom()`. The bottom method has a try block from which an exception is thrown. Different configurations handle the exception differently. [Figure 11.5](#) shows the Spread Around configuration, with an `AWTEException` thrown from method `ccc()`.



Figure 11.5: Advanced Exception Lab reconfigured

Try all the configurations, and be sure that the exception handling makes sense to you in all cases.

Checked Exceptions and Stack Traces

You have already looked at stack traces in the context of runtime exceptions. Checked exceptions also have stack traces. However, you usually don't see the trace because you have to catch checked exceptions. If you want to see a stack trace from a checked exception, you can call the `printStackTrace()` method:

```
try
{
    getNumEnvelopesInStock();
}
catch (ConnectException cx)
{
    System.out.println("Stress!");
    cx.printStackTrace();
}
```

If an exception is thrown, this code will print the "Stress" message, followed by the exception's stack trace. This is extremely useful during development. However, before you ship your code to paying customers, you might want to delete the `printStackTrace()` calls. Paying customers might not want that much information.

Throwing Checked Exceptions

If you're writing a method that throws exceptions, you have to decide which exception type to throw. You have three options:

- Throw `Exception`, as in the examples in the first half of this chapter.
- Throw a subclass of `Exception` from the core Java classes.
- Throw your own custom subclass.

The first option is not realistic. Your code will work, but throwing `Exception` doesn't tell anybody else who reads your code anything about the nature of the exceptional condition. Also, you may need to call your method in a try block that calls other methods that throw. If your method throws `Exception`, it may be difficult or impossible to write a decent set of catch blocks. This is especially true if the try block calls more than one method that throws `Exception` when it could have thrown a more specific type. Your code is always most robust when you throw exceptions that are as specific as possible.

Your second option is to throw a preexisting exception type chosen from the core Java classes. This is easy when you know how to explore the core Java packages and discover the names and behaviors of the many classes they provide. You will learn how to do this in the [next chapter](#). For now, be aware that it's important to choose the most accurate and informative exception name you can find. Most existing types have very long and informative names.

Unfortunately, a lot of programmers always throw `IOException`, even when the problem has nothing to do with Input/Output. This is a bad habit. The rationale seems to be that, out of all the checked subclasses of `Exception`, `IOException` has the shortest name. Please don't yield to this temptation.

Once you decide on an exception type, you construct and throw just as you saw earlier in this chapter, when you constructed and threw `Exception` (which, as you now understand, you should never do). All exception subclasses in the core Java packages have two forms of constructors: a no-arguments version, and a version that takes a text message as an argument. It is always better to use the second form. Be sure to compose a message that is both accurate and helpful.

For example, earlier in this chapter you saw code that called a hypothetical method called `getNumEnvelopesInStock()`, which threw `ConnectException`. Put yourself in the shoes of the person who wrote that method. He might have done something like the following, assuming he had a method called `connectionOK()` that returned `true` if the connection to the database server was sound:

```
public int getNumEnvelopesInStock()
    throws ConnectException
{
    if (connectionOK() == false)
        throw new ConnectException();

    // Get & return # of envelopes remaining.
    . . .
}
```

However, it would be more informative to include a message in the exception. You might pass something like the following into the constructor: "Couldn't get # of envelopes from remote db." Then any catch block that caught your exception could call `getMessage()` on it, printing out the result if appropriate.

What should you do if there's no appropriately named exception subclass in the core packages? You have to fall back on your third option, which is to create your own class. To do this, first decide if you should create a checked exception or a runtime exception. In other words, does your exception represent an unavoidable hazard of existence, or is it a programming error that should be fixed? Most often you will create a checked exception. Next, choose a name. The name should end with `Exception`, because that's what other people expect. For example, you might decide that if the `getNumEnvelopesInStock()` method can't connect to its remote database, it should throw a custom exception type. A plausible name would be `RemoteEnvelopeCountException`. The name says that the class is definitely an exception, and that both remote access and the envelope count are involved.

Having chosen a name, next you have to decide on a superclass. A checked exception should extend `Exception`, `IOException`, or some other checked exception type. In general, extend `IOException` or one of its many subclasses if the exceptional condition you want to represent involves input or output. Otherwise, extend `Exception`. In the rare case when you want to create a runtime exception, extend `RuntimeException`. In this example, `RemoteEnvelopeCountException` will be a subclass of `ConnectException`, since the problem stems from an inability to connect to the remote machine that owns the database.

Your custom class does not need any data or methods. It will inherit everything it needs. All you have to do is create constructors. A custom exception should have both constructor versions. Here is the source for `RemoteEnvelopeCountException`, in its entirety:

```
import java.net.*;

class RemoteEnvelopeCountException
    extends ConnectException
{
    RemoteEnvelopeCountException() { }

    RemoteEnvelopeCountException(String s)
    {
        super(s);
    }
}
```

The import line is required because the `ConnectException` superclass lives in the `java.net` package, which is one of the core Java packages that you'll see in the [next chapter](#). The first constructor is a no-arguments constructor that seems to do nothing (but remember the chain of construction from [Chapter 8, "Inheritance"](#)). The second constructor takes a text message, which is passed to the superclass constructor.

Custom exceptions are thrown and caught just like standard types, so you could call `getNumEnvelopesInStock()` like this:

```
try
{
    int n = getNumEnvelopesInStock();
    System.out.println(n + " envelopes remaining in stock.");
}
catch (RemoteEnvelopeCountException recx)
{
    System.out.println("Stress!");
    System.out.println(recx.getMessage());
}
```

Generally, it's better to use an existing exception type if you can find one whose name accurately and helpfully describes the exceptional condition. However, if no such class exists, creating a custom class is good programming style.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. What happens when you run a program that creates an array of ints and then sets the value of an array component whose index is greater than the length of the array?
2. What happens when you run a program that creates an array of ints whose length is less than zero?
3. What happens when you run a program that prints out the result of dividing a non-zero int by zero?
4. Write a program with a try block that just prints out a message. After the try block, add a catch block that catches `java.io.IOException` (which obviously is not thrown by the try block). Does the code compile? If it compiles, what happens when it runs?
5. Suppose a try block throws many different subclasses of `IOException` (and no other exception types). Suppose you want to catch a few specific subclass types, such as `PrinterIOException` or `ConnectException`. All other exception types should be caught in a safety-net block. Your safety-net block can catch `IOException` or [Exception](#). The code will produce the same behavior either way, but the "Catch Blocks and instanceof" section of this chapter says that it's better to use `IOException`. Speculate on why this is true.
6. What three decisions do you have to make when creating a custom exception subclass?

Chapter 12: The Core Java Packages and Classes

Overview

At this point in your Java education, you have a solid foundation in the essential language. There are some Java features that have not been presented in this book, and won't be, but for the most part you know what you need to know to create programs consisting of classes and interfaces.

The remainder of this book will look at a large number of classes that have been written on your behalf and that you can incorporate into your code. You can use them freely. Moreover, since they are downloaded (along with the compiler and the JVM) whenever anyone downloads Java, you already have them, and you can safely assume that anyone who uses your code has the same set of classes.

We cannot possibly present them all in this book. There are well over a thousand core classes and interfaces, and many of them have specialized functionality that is of interest only to people with the same specialization. Instead of providing an exhaustive survey, we will just introduce the most important classes. Then we will show you how to learn all about the more specialized classes and interfaces. By the end of this chapter, you will have the same fundamental tools as any other Java programmer:

- An understanding of the language.
- A knowledge of certain core classes and interfaces.
- The ability to learn other core classes and interfaces as needed.
- The ability to create your own classes and interfaces, when the supplied ones don't address your needs.

The last item implies that you should use existing code wherever possible. This approach has several powerful benefits:

- The code in the core packages has been thoroughly tested.
- The code in the core packages is available immediately.
- The code in the core packages was developed at somebody else's cost (both time and money).

These benefits are offset by the principle that using an existing class to achieve an inappropriate result is generally more expensive than developing appropriate code from scratch. So a good rule of thumb is: *Use core code when you can, and develop when you must.*

The API Pages

There are more than 1,000 core Java classes, and every one of them has been described in detail by the Java creators. Unlike a lot of manufacturers' technical specifications, these descriptions are well-written, accurate, and helpful. They are provided as a set of interconnected HTML pages that you can download to your hard drive and view with the Web browser of your choice. Like Java itself, they are freely downloadable. If you have not already done so, please download them before continuing with this chapter. You can find instructions on how to do this in [Appendix A](#).

The API pages do a fine job of presenting each class in detail. In fact, they do such a good job that there is no need to duplicate their effort in this chapter. Instead, we will begin by showing you how to use the API pages. After that, we will mostly give you just an overview of the classes and methods, while encouraging you to use your new API skills to look up details when you need them.

Digression: A Personal Anecdote

Quite a while ago, before the birth of the World Wide Web, I worked for a company that made computers. These computers used a programming language that was a bit like C, but it was object-oriented. In addition to the language, there were a number of classes that supported I/O, graphical user interfaces, math, and so on. If you saw it today, you would probably be reminded of Java, but with fewer supplied classes.

There were several dozen of these classes. Their documentation consisted of two manuals that listed the classes in alphabetical order. For each class, the manuals listed the inheritance hierarchy, the data, and the methods.

Those of us who wrote programs for this system each had our own copy of the manuals. You could tell how long someone had been working there by the shape their manuals were in. Those books took a beating. We were always flipping back and forth. If I wanted to remind myself what a certain method of a certain class did, I might find that the method wasn't explained where I expected an explanation, because the class I was reading about inherited the method from its superclass. So I would look up the superclass (which might be in the other volume), and I would see that it returned an object reference, and I would have to look up that object's explanation because I hadn't seen it before.

I flipped a lot of pages because a lot of information for one class was (quite rightly) presented in the description of a different class. For example:

- The class's superclass
- The type of a non-primitive variable
- The type of a non-primitive method argument
- The type of a non-primitive method return value
- Any class, method, or variable mentioned for any reason in the description I was reading

A set of HTML documents would have eliminated all that page-turning. The only problem was, this was 1988 and there was no HTML. The Web was just a glimmer in the eyes of a few people in Switzerland, and hypertext was an idea that wasn't discussed much outside of universities. We programmers would often wonder, "Couldn't we fix it so I could read all this on my screen, and somehow click on names of classes and data and methods to read their explanations?" But we didn't invent the World Wide Web.

Fast-forward to right now. We don't have a few *dozen* classes; we have more than a *thousand*. If you printed their explanations in books, how many yards of shelf space would be required? How long would it take to look something up? Fortunately, it's all in HTML files, and fortunately, someone invented the Web. But it wasn't me.

Starting at the Top

To get the most out of this section, read it in front of your computer. You will be invited to look at various API pages. If you haven't yet downloaded and installed them, do it now!

[Appendix A](#) suggested that you download the Java documentation into `j2sdk1.4.1_02\docs` (depending on your operating system, the path separator might be a forward slash). In the `docs` directory, there is a directory called `api`. Display the contents of that directory. Double-click on the icon for `index.html`. Your browser will appear, displaying a fairly large page with 3 frames. This is the index. It is your entry point into the dozens of millions of bytes of information that are your electronic documentation.

The API pages are copyrighted, so we can't show you a picture of the index. Its structure is shown in [Figure 12.1](#).



Figure 12.1: Structure of the API index

The index has 3 frames:

- The packages frame
- The classes frame
- The details frame

Look at the packages frame in the upper-left corner of your browser. It is a list of all the core Java packages. Subpackages are also listed. Notice that the 2nd package in the list is `java.awt`, which is followed by its 10 subpackages.

The remainder of the left edge of the page is occupied by the classes frame. Initially, all the classes of all the core packages are displayed. When you click on an individual package in the packages frame, the classes frame displays the contents of the selected package. [Figure 12.2](#) shows the structure of the classes frame.

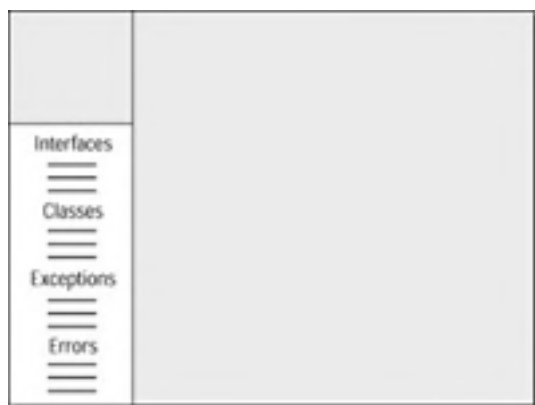


Figure 12.2: Structure of the classes frame

The classes frame shows all the interfaces, classes, exceptions, and errors of the selected package. (If you haven't yet selected a package, or if you select All Classes in the packages frame, you see everything for all packages.) Each interface, class, exception, and error is a link.

We haven't discussed errors in this book. They are like exceptions, but they indicate something deeply wrong with a program. As a programmer, you should avoid throwing or catching errors. So for the remainder of this book, we will continue to ignore their existence.

Try it. In the packages frame, click on `java.awt`. (We will spend the last three chapters of this book learning how to use this package.) The classes frame shows that `java.awt` has a large number of interfaces, a very large number of classes, a few exceptions, and one error.

Now go back to the packages frame, scroll down a bit, and click on `java.lang`. Click on the link for the `Boolean` class, which is the first link in the list of classes. The details frame displays a complete explanation of the class.

The class description is quite long, even for a simple class like `java.lang.Boolean`. It is divided into 3 sections, as shown in [Figure 12.3](#).

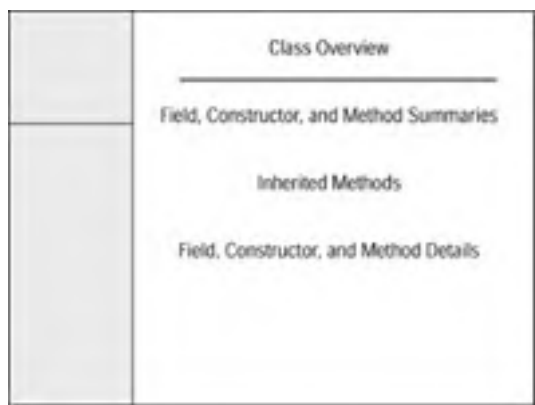


Figure 12.3: Class description

The class overview presents the class name, its inheritance hierarchy, and a text description. All elements of the inheritance hierarchy are links, so it's easy to look up a class's superclass.

The field, constructor, and method summaries are the really useful part of this frame. [Figure 12.4](#) shows their structure.

Field Summary	
data type	field name description
data type	field name description

Constructor Summary	
constructor signature	constructordescription
constructor signature	constructordescription

Method Summary	
return type	method signature description
return type	method signature description

Figure 12.4: Field/constructor/ method summaries

Each field, constructor, and method is listed alphabetically, along with a brief description. To see a more detailed description, click on the name of the field, constructor, or method. The name is a link to the position, further down on the same page, of the detailed description. Any class name in the detailed description is a link to the API page for that class. Try it. In the method summary section, click on the link for `toString()`. You see an explanation that is somewhat deeper than that brief one that appeared in the summary. Any exceptions thrown by any constructor or method are listed in the detailed description.

Any field, constructor, method, or class mentioned anywhere in the details frame is itself a link. The return type of the `toString()` method is `String`. Click on one of the occurrences of this word. The details frame displays the description of the `java.lang.String` class.

Between the summary and detail sections is a short section that lists all the methods that the class inherits from all its parent superclasses.

Typically, a session with the API pages goes like this. You want to look up a particular method of a particular class, for any of the same reasons you would look up a word in a dictionary. You want to know:

- How to spell it.
- How to use it.
- What it means.

For a method, you probably want to know one of the following:

- Its spelling.
- Its return type.
- Its argument list.
- What it does.
- What exceptions it throws.

You begin your session by scrolling through the packages frame until you find the right package. You click on it, so that the classes frame displays the contents of the package. You scroll through the classes frame until you find the class you want. You click on the class link to make the details frame display the class description.

Now you scroll through the alphabetical list of methods until you find the one you want. The summary information might be enough. If not, you click on the method name to view the detailed description.

If you're looking for the method's return type or argument list, you might find yourself looking at the name of a class that you don't recognize. No problem. Click on the class name (it's a link) and read its API page.

The API pages contain more information than is presented here, but this is enough to get you going. If you are curious about the additional API information, a good place to start is the very top of any class description page. Click on the Use, Tree, or Index link.

Deprecated Methods

Occasionally, an API page might tell you that a certain method is *deprecated*. A deprecated method is one that was introduced in an early revision of Java, and since then has been replaced with something more robust, modern, bug-free, or trustworthy. You are strongly cautioned not to use anything that is deprecated. Sun reserves the right to remove anything deprecated from future revisions.

Ordinarily, Java is *backward-compatible*. This means that if you write Java code that compiles successfully and runs correctly with the current revision of Java, your code will still compile successfully and run with the same behavior as before in any future revision of Java. But if you use deprecated methods, you no longer get backward-compatibility. If one of the methods you call has been removed, your code will no longer work.

Team LiB

← PREVIOUS NEXT →

The *java.lang* Package

The core Java classes fall into three broad categories. There are classes that support specific tasks, such as database access or graphical user interface (*GUI*) creation. These classes are organized into packages. For example, database support is in the `java.sql` package, while GUI infrastructure is in the `java.awt` package and its many subpackages. Another category is classes that are generally useful, no matter what kind of program you are writing. Most of these appear in the `java.util` package. The third category is classes that are essential to the operation of any program. These are to be found in the `java.lang` package. Let's begin with a look at a few of them.

The classes and interfaces in `java.lang` are so important that they are imported in all source code automatically. It is as if the compiler inserted the following line into any source:

```
import java.lang.*;
```

We will not be looking at all the classes in `java.lang`. Again, the purpose of this book is not to tell you everything there is to know about every class in the package. Instead, we will just look at a few of the most important classes. Since you know how to read API pages, you know how to find out about the others.

The *java.lang.String* Class

We will start with `String`. You have been aware of this class ever since [Chapter 2](#), when you first looked at applications and saw the following:

```
public static void main(String[] args)
{
    . . .
}
```

Now you know that this is a method declaration, and no doubt you've guessed that the method takes a single argument whose type is an array of something called `String`.

The `String` class contains an ordered sequence of characters, representing a piece of text. The text that an instance contains is specified as a constructor argument. The class is *immutable*. This means that after an instance is constructed, its contents cannot be changed. So if an instance of `String` initially contains the text "Click here to select a color", it will contain that text throughout its lifetime.

This class is unique, in that there are two ways to create an instance. One way, of course, is to call a constructor. The second way, which is unique to the `String` class, is to use a *literal string*. A literal string is text enclosed in double-quotes, like this:

```
"I am a literal string."
```

When the compiler encounters a literal string, it generates code that creates an instance of `String` to represent the text in quotes. (Actually, the situation is a bit more complicated than that, but we don't need to go into detail here.) We have often used code with the following format:

```
System.out.println("value is " + x);
```

Now you know that the text between the quotes is a literal string. Later in this chapter, you will see what is really going on when the literal string is added to `x`. For now, you know that whatever else might be happening in the line of code, execution involves the creation of an instance of `String` that represents the text in quotes.

The shortest literal string is

```
""
```

This string has zero characters, but it is still an object that exists, and you can call any methods of the `String` class on it. It is called the *empty string*.

The easiest way to create a `String` instance that contains a particular run of text is like this:

```
String s;
s = "To be, or not to be";
```

Or simply:

```
String s = "To be, or not to be";
```

There are 11 different versions of the `String` constructor. Here we will only look at one of them. (Later in this chapter you will learn how to look for information about the rest, so you will be able to choose the best one for any situation.) The simplest constructor is

```
public String(String s)
```

This constructor takes a single argument, which is a reference to another string. The new object is an exact replica of the argument. Even though this is the simplest form of the `String` constructor, it isn't often used because you have the option of using a literal string instead. For example, the following two lines are (almost) equivalent:

```
String s = new String("abcde");
String s = "abcde";
```

The second version is obviously easier to type.

The `String` class has a large number of methods. Here we will present a few of the more interesting ones. We start with `toUpperCase()` and `toLowerCase()`. The `toUpperCase()` method converts all lowercase characters in a string to uppercase. The `toLowerCase()` method converts all uppercase characters in a string to lowercase. Here is an example:

```
String s = "Mm";
String upper = s.toUpperCase();
String lower = s.toLowerCase();
System.out.println(upper);
System.out.println(lower);
```

The output of this code is

```
MM
Mm
```

These methods, and indeed all `String` methods that seem to modify a string, do not change the original string. That would be impossible, because strings are immutable. After they are constructed, they cannot be changed. Instead, `toUpperCase()` and `toLowerCase()` create and return a new string object. To prove this, you can modify the code example:

```
String s = "Mm";
String upper = s.toUpperCase();
String lower = s.toLowerCase();
System.out.println(upper);
System.out.println(lower);
System.out.println(s);
```

The output of this code is

```
MM
mm
Mm
```

Notice the last line, which prints out the unscathed original string.

The situation can get confusing when a single reference is reassigned, as in the following example:

```
String s = new String("UPPER : lower");
System.out.println("BEFORE: " + s);
s = s.toUpperCase();
System.out.println("AFTER: " + s);
```

Here the single reference `s` refers first to the original string, and a little later to the new uppercase string. It *looks like* there is only a single string object involved, especially when you look at the output:

```
BEFORE: UPPER : lower
AFTER:  UPPER : LOWER
```

But there are actually two objects, although there is only one reference. When the reference is reassigned (`s = s.toUpperCase()`), the original string object might get garbage- collected. This would happen if there were no other references to the original object.

This might seem overly complicated when all you wanted to do was convert a string to upper- or lowercase, but it is always important to bear in mind the difference between references and objects, and to know exactly what references are pointing to what objects at every point in your code.

The `StringLab` animated illustration demonstrates strings in moving pictures. To run the animation, type `java strings.StringLab`. You see a window with two code statements, as shown in [Figure 12.5](#).

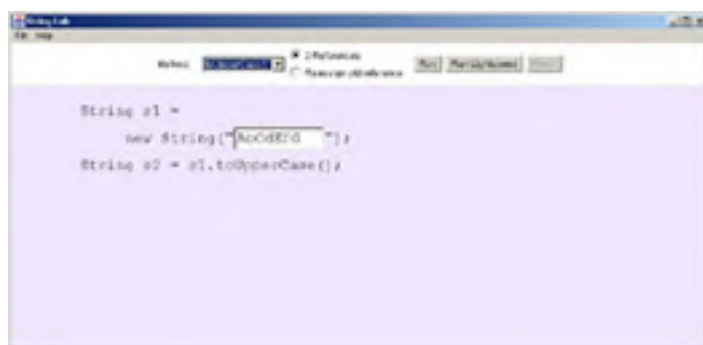


Figure 12.5: StringLab

The first statement creates a new instance of `String`. You can type anything you like to provide the text. The second statement calls a method, `toUpperCase()` or `toLowerCase()`, on the string. Use the controls at the top of the window to choose the method that will be called. You can also use the controls to choose between using two references or reassigning a single reference. (Notice how the code in the main window changes when you change this option.) As usual, click `Run` to see the animation, or click `Run Lightspeed` to skip the animation and see the final result. Be sure to watch the full animation with `Reassign Old Reference` selected so you can see the original string being garbage-collected.

[Figure 12.6](#) shows the result of converting to uppercase and using two references. [Figure 12.7](#) shows the result of converting to lowercase and reassigning the reference.



Figure 12.6: StringLab: uppercase, 2 references



Figure 12.7: StringLab: lowercase, 1 reference

Now that you understand strings and methods, we can move quickly through a list of other `String` methods:

- `trim()` Removes blank spaces from the start and end (but not the middle) of the executing string object.
- `substring(int n)` Returns a portion of the executing string object. The substring consists of the run of characters beginning at position `n` (where 0 is the first character) and ending at the end of the executing string object.
- `concat(String s)` Appends `s` to the executing string object.

The return types are all `String`. These descriptions are deliberately brief. Detailed explanations are available to you in the API pages.

Here are some `String` methods that return information about the executing string object. The return types vary, so they are included in the list:

- `boolean equals(String s)` Returns `true` if `s` and the executing string object contain identical text.
- `boolean equalsIgnoreCase(String s)` Returns `true` if `s` and the executing string object contain identical text, ignoring uppercase and lowercase distinctions.
- `char charAt(int n)` Returns the `n`th character in the executing string object.
- `int length()` Returns the length of the executing string object.
- `boolean startsWith(String s)` Returns `true` if the executing string object begins with string `s`.

Here is a method whose argument is a string. The method prints out every character of the argument on its own line:

```
void printChars(String s)
{
    int length = s.length();
    for (int i=0; i<length; i++)
    {
        char c = s.charAt(i);
        System.out.println("Char #" + i + " is " + c);
    }
}
```

Note the use of the `length()` and `charAt()` methods. Here is the output when the method is called with argument `Alligator`:

```
Char #0 is A
Char #1 is l
Char #2 is l
Char #3 is i
Char #4 is g
Char #5 is a
Char #6 is t
Char #7 is o
Char #8 is r
```

The only tricky method presented here is `equals()`. It is easy to understand what it does, provided you don't get misled by the name. The `equals()` method does not check if the argument and the executing string object are the same object.

Instead, it checks whether the two objects both represent identical sequences of characters. To check if the argument and the executing string are the same object, use the `==` operator, as demonstrated in the following example:

```
String s1 = new String("aa");
String s2 = new String("aa");
String s3 = s2;
if (s1.equals(s2))
    System.out.println("s1.equals(s2): YES");
if (s1 == s2)
    System.out.println("s1 == s2: YES");
if (s2 == s3)
    System.out.println("s2 == s3: YES");
```

The output is

```
s1.equals(s2): YES
s2 == s3: YES
```

Figure 12.8 shows the references and objects of this example.

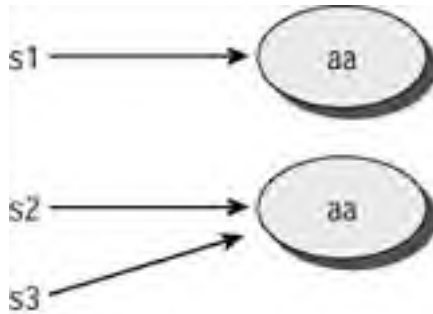


Figure 12.8: String references and objects

We say that the `==` operator checks for *reference equality*, which means it checks if two references point to the same object. The `equals()` method checks for *object equality*, which means it checks if two objects contain equal data. This distinction is very important in object-oriented programming.

Command-Line Arguments

Every Java application has a main method that begins like this:

```
public static void main(String[] args) . . .
```

Of course, you can call the method argument anything you like, but `args` is the conventional name. The array contains the application's *command-line arguments*. These are everything the user has typed into the command line that invoked the application, except for the following:

- `java`
- The application class name
- Any arguments for the JVM

So if you have an application class called `database.Backup`, and you run it by typing `java database.Backup network local greebo 1234`, the `args` array will look like Figure 12.9.

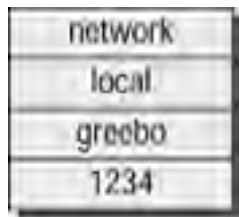


Figure 12.9: Command-line arguments

Note that the last component in the array is a string. It is not a number, even though it looks like one because it consists entirely of digit characters. Later you will learn how to convert an all-digit string into an int.

One job of the `main()` method is to check the command-line arguments and take appropriate action. For example, a `main()` method might have the following code:

```
public static void main(String[] args)
{
    for (int i=0; i< args.length; i++)
    {
        if (args [i].equals("help"))
        {
            printHelpMessage();
            break;
        }
    }
}
```

We assume the existence of a `printHelpMessage()` method that prints out an explanatory message on how to use the program. The code scans the `args` array. If the array contains any occurrence of "help" (that is, if someone typed "help" anywhere on the command line), the explanatory message is printed.

Some command-line arguments are intended for the JVM and do not get passed into the application. To see a list of JVM arguments, type `java -help`. One such argument is `-verbose`. When you run a program with this argument, you get output from the JVM about such activities as class loading. If you invoke an application by typing `java -verbose database .Backup network local greebo 1234`, the `-verbose` argument will be consumed by the JVM and won't be passed on to the application. So the application's `args` array will still be as shown in [Figure 12.9](#), but the output will include the verbose messages from the JVM.

The `java.lang.Object` Class

The `Object` class is the ultimate superclass of all other Java classes. It has about a dozen methods, many of which support advanced functionality that is beyond the scope of this book. But it does have two methods that everyone should know about: `equals()` and `toString()`. Since every other class extends `Object` (directly or indirectly), every other class inherits these methods. You are not likely to create instances of `Object`, but you are very likely to use the `equals()` and `toString()` methods that other classes inherit or override.

The first method we will look at is `equals()`. You already know about the implementation provided by the `String` class, and that it checks for object equality and not reference equality. This distinction is maintained throughout the core Java classes: Any implementation of `equals()` in any core class checks for object equality rather than reference equality. So `thisRef.equals(thatRef)` will return `true` if the two references point to objects that contain equal variables. (Of course, if the two references point to the same object, the call will also return `true`.)

For example, the `java.awt.Point` class represents a point in two-dimensional space. It has variables called `x` and `y`, which hold the horizontal and vertical location of the point. If `p1` and `p2` are both references to instances of this class, you can check for object equality by calling either

```
if (p1.equals(p2)) ...
```

or

```
if (p2.equals(p1)) ...
```

The `equals()` method of class `Point` returns `true` if `p1.x` equals `p2.x` and `p1.y` equals `p2.y`.

Not all core classes provide their own implementations of `equals()`. If you want to know if a class you are interested in provides an implementation, you should look at the class's API page. Beware of classes whose API pages do not document an `equals()` method. The class might inherit the method from `Object`, and that version is not very useful.

Now we will turn to the extremely useful `toString()` method. It is public, it returns a `String`, and it has no arguments, so its declaration looks like this:

```
public String toString()
```

This method is intended to print out a useful message that includes information about the values of the executing object's variables. The version provided by the `Object` class isn't very informative. In fact, it's downright cryptic. But every one of the core Java classes overrides `toString()` with a version that provides useful information.

For example, there is a class called `java.awt.Color` that represents a color. The class has `int` variables called `red`, `green`, and `blue`, which contain the amounts of red, green, and blue light that make up the represented color. Their values can range from 0 through 255, and they are specified in the class's constructor.

Suppose you have a long intricate program with an instance of `Color`, referenced by variable `foreground`, that doesn't look right. (Colors are used extensively in visual programming, which we will look at in the last 3 chapters of this book.) It would be helpful if you knew exactly what the red, green, and blue components of the problematic color are. That information might lead you to the source of the trouble. Thanks to `toString()`, you can easily create a line of [debug code](#) that tells you what is going on inside your program. Always delete debug code after it has served your purpose. Otherwise it will accumulate, and your program will emit lots of information that is no longer helpful.

```
Here is a debug line that prints out the puzzling color:
System.out.println("Weird color is " +
    foreground.toString());
```

The output looks something like this:

```
Weird color is java.awt.Color[r=100,g=255,b=0]
```

The output uses abbreviations instead of `red`, `green`, and `blue`, but it's clear what their values are. Now you can determine which of them are wrong and look at the code that calculates the corresponding value that is passed into the `Color` constructor.

There is an easier way to print the `toString` value of the color, or of any other object. This brings us to the topic of [string concatenation](#). "Concatenation" is another of those five-syllable words. It just means joining strings consecutively, one after another (after another, after another...). You have already used concatenation extensively, whenever you did something like

```
System.out.println("size is " + size +  
                  "and weight is " + weight);
```

Now it's time to see what's really going on between those parentheses. Look at those plus signs. Obviously they mean something other than addition. In Java, plus signs have a double meaning:

- When both items next to a plus sign are numeric, the plus sign means addition.
- When one or both items next to a plus sign are references to `strings`, the plus sign means string concatenation.

In the second context, if one of the items next to the plus sign is a string, the other item can be *anything!* It can be a primitive, another string reference, or a reference to an object of any other class. The other item is converted to a string according to the rules shown in [Table 12.1](#).

Table 12.1: String Concatenation Conversion Rules

Type	Conversion Rule
<code>boolean</code>	"true" or "false"
Primitive other than <code>boolean</code>	A reasonable string representation
<code>String</code>	The string
Object reference other than <code>String</code>	Call <code>toString()</code> on the reference

The last entry in the table means that the line

```
System.out.println("Weird color is " +  
                  foreground.toString());
```

is equivalent to

```
System.out.println("Weird color is " +  
                  foreground);
```

In other words, when you're doing concatenation, you never need to type `.toString()`.

The ConcatLab animated illustration shows concatenation in action. Run the program by typing `java concat.ConcatLab`. You will see a window with three lines of code, as shown in [Figure 12.10](#).



Figure 12.10: ConcatLab

The first statement creates an instance of `java.awt.Color`. The constructor's arguments are the three primary color values (red/green/blue) that constitute the color. They must be in the range 0-255. You can type in any valid values. Later on, you will see the color they represent.

The second statement creates an instance of a class called `Point3D`, which is not part of the core Java classes. To see the (very simple) source for `Point3D`, click on the Edit `Point3D` button. You will see the display shown in [Figure 12.11](#).



Figure 12.11: ConcatLab's `Point3D` class

The class represents a point in 3-D space, with x, y, and z coordinates. You will see a version of toString() that returns a reasonable string representation. You can edit this code. Type any text you want into the text fields. When you're finished, click on OK.

The third statement on the main window says

```
String s = "c is " + c + " and p is " + p;
```

Click on the Run button to run the animation, which shows how the four parts of the concatenated string are made. [Figure 12.12](#) shows the result of running the animation, after some custom configuration.

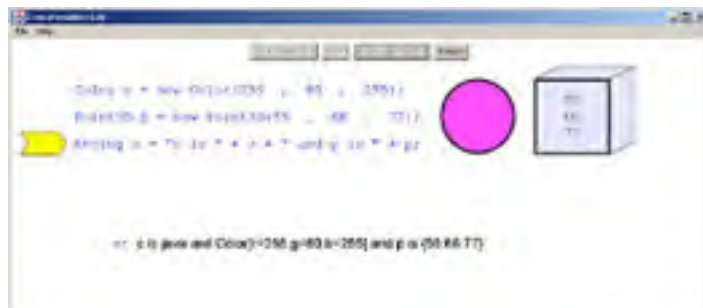


Figure 12.12: ConcatLab's Point3D class

The `java.lang.Integer` Class, and other Wrappers

The `java.lang` package has eight very simple classes called *wrappers*. A wrapper is a class whose data is a single primitive value. In other words, the primitive is "wrapped up" inside an object. The wrapper classes have names that are very similar to the corresponding primitive names. In some cases, the names are identical except for the first letter, which is always uppercase for class names and lowercase for primitive names. [Table 12.2](#) shows the wrapper class names.

Table 12.2: Wrapper Class Names

Primitive	Wrapper
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

The wrapper classes are immutable. This means that, just as with strings, the data contained in an instance doesn't change after the instance is created. The contained data is passed into the constructor, as shown below:

```
Boolean boo = new Boolean(true);  
Character cha = new Character('L');
```

```
byte b = 5;  
Byte bye = new Byte(b);  
short s = 10;  
Short sho = new Short(s);  
int i = 9999;  
Integer inty = new Integer(i);  
long n = 2222222;  
Long lonny = new Long(n);  
float f = 3.14159f;  
Float flo = new Float(f);  
double d = 1.2e200;  
Double dubby = new Double(d);
```

It might not be clear why these classes would ever be useful. You'll find out why when you learn about the `java.util` class `a` little later on. But for now, be aware that the wrapper classes all have static methods that are useful for translating strings into primitives. For example, class `Integer` has a `parseInt(String s)` method that translates a string to an int. If the string does not represent a number, the method throws `NumberFormatException`.

The following code is an application that translates its first classes to an int, multiplies that value by 39, and then prints out the result:

```
public class X39  
{  
    public static void main(String[] args)  
    {
```

```
if (args.length == 0)
{
    System.out.println("Please supply a number.");
}
else
{
    try
    {
        int n = Integer.parseInt(args[0]);
        int times39 = n * 39;
        System.out.println(args[0] + " * 39 = " +
            times39);
    }
    catch (NumberFormatException x)
    {
        System.out.println("That's not a number!");
    }
}
}
```

Note that this code doesn't create an instance of `Integer`. Instead, it calls a static method of `Integer` (in the first line of the try block) to convert the string in `args[0]` to an int. Code like this is frequently seen near the beginning of `main()` methods. Generally, an application that has numeric command-line arguments can't get very far until it converts the argument strings to numeric primitives.

The `java.lang.System` Class

The `java.lang.System` class contains a hodgepodge of methods, most of which involve advanced functionality related to the JVM. This brief section will only cover one of the class's methods. First, let's take a moment to look at two of its static variables: `out` and `in`.

You have already used `System.out` extensively. Whenever you used

```
System.out.println(...);
```

You were making a call to the `println()` method of the `System.out` object.

The `println()` method is heavily overloaded. All of the versions of the method take a single argument. One version, which you have been using throughout this book, takes a string argument. The string is printed out, followed by a *newline* character. The newline character is not displayed. Instead, it moves the cursor position to the beginning of the next line.

Other versions of `println()` take args that are bytes, shorts, booleans, and so on. These versions convert their arguments to strings and then print them out, followed by a newline character.

Perhaps the most commonly used method of `java.lang.System` is `exit()`. This method causes the JVM to terminate, thus ending the current application immediately. The method takes an int argument called the *exit code*. Typically, 0 is used to indicate a normal termination, while a non-zero value indicates that termination was caused by an error condition.

Some operating systems are able to run sequences of programs, where the exit code of one program is used to control the operation of the next program. This is highly system-dependent and not relevant to an introductory Java book, but you need to know what exit status codes are because you need to pass an argument into every `System.exit()` call. You won't go wrong if you use 0 to mean normal termination and a small non-zero value to mean abnormal termination.

The following code is a rewrite of the previous example, using `System.exit()` to terminate execution.

```
public class X39RevB
{
    public static void main(String[] args)
    {
        if (args.length == 0)
        {
            System.out.println("Please supply a number.");
            System.exit(1); // Non-zero exit code
        }
        else
        {
            try
            {
                int n = Integer.parseInt(args[0]);
                int times39 = n * 39;
                System.out.println(args[0] + " * 39 = " +
                    times39);

                System.exit(0);
            }
            catch (NumberFormatException x)
            {
                System.out.println("That's not a number!");
                System.exit(2);
            }
        }
    }
}
```

The `System.exit(0)` call at the end of the try block isn't actually necessary. When `main()` finishes, the JVM shuts down anyway, with an exit code of 0.

The `java.lang.Math` Class

The `java.lang.Math` class is the least object-oriented of all the core Java classes. All of its methods are static, so you never need to create an instance. In fact, you aren't *allowed* to create an instance (see Exercise 3 for details).

The class has dozens of methods. Instead of listing them all, here's a short sampler. Consult the API page for the complete list.

```
int min(int a, int b) Returns the lesser of a and b
int max(int a, int b) Returns the greater of a and b
double sin(double angle) Computes the sine of an angle
double cos(double angle) Computes the cosine of an angle
double tan(double angle) Computes the tangent of an angle
double pow(double a, double b) Returns a raised to the power of b
double random() Returns a random number that is >=0 and <1
```

And so on. All trigonometry methods use radians, not degrees, for expressing angles. All methods that do intense calculation have return types of `double`.

The `random()` method can be used to generate a random double in any range. For example, to generate a random double that is ≥ 0 and < 30 , just use `30 * Math.random()`. To generate a random double that is ≥ 10 and < 40 , just use `10 + (30*Math.random())`.

The following code generates 100 random numbers that are ≥ 0 and < 50 . Then it computes the area of a circle whose radius is the random number. The code keeps track of, and prints out, the largest area:

```
public class RandomAreas
{
    public static void main(String[] args)
    {
        double maxArea = 0;
        for (int i=0; i<100; i++)
        {
            double radius = 50 * Math.random();
            double area = 3.24159 * radius * radius;
            if (area > maxArea)
                maxArea = area;
        }
        System.out.println("Biggest area = " + maxArea);
    }
}
```

The `lang` class defines two public final static double variables, `PI` and `E`. They contain very precise values for these mathematical constants, accurate to 20 digits to the right of the decimal point. So you never need to memorize, look up, or type in either of these values. That's a good thing, because you might accidentally type in a wrong value that could throw off all your subsequent calculations. In fact, that's what happened in this example. In the line that begins `double area =`, the `2` should be a `1`. A better line would be

```
double area = Math.PI * radius * radius;
```

You can make the code a little shorter by replacing these lines:

```
if (area > maxArea)
    maxArea = area;
```

With this single line:

```
maxArea = Math.max(area, maxArea);
```

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. In the beginning of this chapter you learned that a good rule of thumb is to use core code when you can and develop original code when you must. Because Java is an object-oriented language, you have a third option, which combines reusing existing code with creating your own. You learned about this option in an earlier chapter. What is it?
2. If you write code that calls a deprecated method of one of the core Java classes, what valuable feature of Java can you no longer rely on?
3. Suppose you are reading someone else's code and you come across the following lines:

```
Stack myStack = new Stack(); // java.util package
myStack.setSize(100);
```

You decide to look up `setSize()` in the APIs. The comment kindly tells you that class `Stack` is in package `java.util`, so you click on `java.util` in the packages frame, and then you click on `Stack` in the classes frame. You find yourself looking at the class description. You scroll down to the method summaries, and you don't see `setSize` anywhere.

How should you proceed?

4. In the section on the `String` class, you learned about the `startsWith(String s)` method, which returns `true` if the executing string object begins with the argument string `s`. It stands to reason that there should be a similar method that tells you whether the executing string object ends with a specified string. Look at the API page for `java.lang.String` and see if such a method exists.
5. What happens when you try to compile and execute the following application?

```
public class Ch12Q5
{
    public String toString()
    {
        return "I am an instance of Ch12Q5.";
    }
    public static void main(String[] args)
    {
        Ch12Q5 thing = new Ch12Q5();
        System.out.println(thing);
    }
}
```

6. What happens when you try to compile and execute the following application?

```
class Ch12Q6
{
    String toString()
    {
        return "I am an instance of Ch12Q6.";
    }
    public static void main(String[] args)
    {
        Ch12Q6 thing = new Ch12Q6();
        System.out.println(thing);
    }
}
```

7. Look up the explanation of the `equals()` method on the API page of class `java.lang.Object`. The explanation is a bit wordy, but see if you can figure out what it does. (Focus on the last sentence, just before the "Parameters" section.) What is the technical term for what the method does? (Hint: It was introduced in this chapter.)
8. You're not allowed to construct an instance of the `java.lang.Math` class. What happens if you try?
9. The following code models the behavior of a familiar piece of equipment that is used in many games throughout the world. What is the piece of equipment?

```
long rand = 1 + Math.round(Math.random() * 5);
```

Chapter 13: File Input and Output

In [Chapter 12](#), you saw a few of the core Java packages and classes. You also learned that creating successful Java programs involves both writing your own code and using preexisting classes. This chapter will cover the fundamentals of reading and writing disk files. It will take advantage of several core classes in the `java.io` package. This will be your first look at making extensive use of core classes.

Files As Sequences of Bytes

In [Chapter 1](#), you saw that a computer's memory is a clump of tiny circuits in which voltages represent 0s and 1s. It doesn't take a degree in electrical engineering to know that when you turn off a circuit's power, the voltages go away. No more 0s, no more 1s.

Disks are like computer memory in the following sense: A disk is a collection of tiny "somethings" that can be in one of two possible states. The surrounding electronics, and the software that controls the surrounding electronics, interpret the two states as representing 0 or 1. With a hard disk, the states are microscopic magnetic fields that can point in either of two directions. With a CD-ROM or DVD, the medium is filled with microscopic regions that either do or do not block light. Aside from the underlying physics, the main difference between disks and memory is that disks remember what is stored in them, even after the power goes off.

To make the rest of this discussion more clear, let's use the term *RAM* to mean ordinary computer memory, as distinct from disks, which are also a kind of memory. RAM is an acronym for Random Access Memory. It's a cool-sounding acronym, but you may be wondering what's so random about RAM. "Random" relates (distantly) to the amount of time it takes to read data out of memory or to write data into memory. It takes exactly the same amount of time (less than one millionth of a second) to read any byte in the circuit. Writing might take slightly longer than reading, but writing any byte takes exactly the same amount of time as writing any other byte. So you can pick any two bytes at random, and they can be read in the same amount of time, or written in the same amount of time, as each other.

Disks are not random access devices. At any moment, some parts of the disk data can be read more quickly than others. This is because the disk is rotating. If you want to read some data, you have to wait until it has rotated into position next to the disk's reading or writing hardware, which does not rotate. If you're lucky, the data will be just about rotated into position. If you're out of luck, the data will have just rotated out of position, and you will have to wait until the disk makes another revolution.

So you see that RAM and disks have very different mechanical and physical properties, but they both can be treated as storing ordered sequences of 0s and 1s.

As with RAM, you think of disks as being organized into bytes, each byte having a unique position. As with RAM, you would find it impossibly limiting if you had to think exclusively in terms of bytes. As with RAM, you use groups of disk bytes to encode higher-level multi-byte information. But unlike RAM, the first step in learning how to do disk input and output is to learn how to read and write pure bytes. That is where we will begin.

Writing and Reading Bytes

Document files don't really contain text. Image files don't really contain pictures. MP3 files don't really contain music, and MPEG files don't really contain movies. They all contain bytes, because all files just contain bytes. The bytes encode information; they are decoded by software appropriate to the encoded content. This is why filename extensions are so important. They tell the computer what decoding software to use. If you take an image file that encodes a really beautiful picture and you change filename extension to .mp3, it's probably going to sound terrible.

All files are sequences of bytes. Before we look at decoding and encoding the information represented by the files, you need to learn how to write and read plain ordinary bytes. You will make extensive use of two of the classes in the `java.io` package:

- `FileOutputStream`
- `FileInputStream`

A file output stream writes bytes to a file; a file input stream reads bytes from a file. Our purpose here is not to present both classes in their entirety. Here you will learn more than enough to be able to use them well. Whenever you want to complete your understanding, you can refer to the API documentation.

Both classes have constructors with `String` arguments, where the string specifies the name of the file. On Windows machines, the *file separator* (the character that goes between elements in a full pathname) is a backslash, and that can lead to problems. So let's begin with a digression on dealing with backslashes.

Backslashes in Filenames

If you want to write bytes to a file in the current working directory called `xyz`, you can construct a file output stream like this:

```
FileOutputStream fos;  
fos = new FileOutputStream("xyz");
```

Of course, you can create a file output stream in a similar way. Ignoring for the moment the issue of what you can actually do with those streams, you have to deal with the question of what happens when you want to specify a full pathname on a Windows system. For example, what if you want to write to a file whose full pathname is `C:my_files\photos\abc`? The following code will not do what you want:

```
FileOutputStream fos;  
fos = new FileOutputStream("C:my_files\photos\abc");
```

Surprisingly, this code will not compile! The compiler error says that there is an invalid escape character, whatever that means.

Actually, the problem has nothing to do with file output streams. It has to do with backslashes in literal strings. You would get the same compilation error if you tried the following:

```
String s = "C:my_files\photos\abc";
```

In [Chapter 2, "Data,"](#) you saw that certain characters (most notably the newline and tab characters) are represented by escape codes, `\n` for newline and `\t` for tab. Those codes can also be embedded in literal strings. For example, the following code prints some numbers, separated by tabs, on two lines:

```
String s = "123\t456\t789\n987\t654\t432";  
System.out.println(s);
```

You can see that the backslash character has special meaning to the Java compiler. In literal strings and chars, backslash means, "Ignore me and treat the next character as a special code." If you just want a simple ordinary backslash in a literal string or char, you have to use a double backslash. For example, to print out the word "hello" followed by a backslash, you have to do the following:

```
System.out.println("Hello\\");
```

Note the second backslash. Only one backslash is printed.

So you can see that

```
String s = "C:my_files\photos\abc";
```

won't compile, because `\p` and `a` are not valid escape codes. It's a good thing they aren't. The following code compiles, but with an unexpected result:

```
String s = "C:my_backup\temporary\news";
```

So if you are writing file access code for a Windows machine, you always have to remember to use double backslashes for file separators, like this:

```
FileOutputStream fos;  
fos = new FileOutputStream("C:my_files\\photos\\abc");
```

Now that you know how to specify filenames, we can move on to writing to files.

Writing Bytes

To create a file full of bytes, you have to do three things:

1. Construct an instance of `FileOutputStream`.
2. Write the bytes.
3. Close the stream.

The following application creates a file called "xyz" in the current directory, writes 10 bytes, and then closes the file:

```
1. import java.io.*;
2.
3. public class Write10Bytes
4. {
5.     public static void main(String[] args)
6.     {
7.         FileOutputStream fos;
8.         fos = new FileOutputStream("xyz");
9.         for (int i=0; i<10; i++)
10.            fos.write(i);
11.        fos.close();
12.    }
13. }
```

Line 8 constructs the output stream. Line 10, which executes 10 times in the `for` loop, writes the bytes. It looks like line 10 actually writes ints, because `i` is an int, but the `write()` method actually only writes the low-order 8 bits of its argument. Line 11 "closes" the stream. *Closing* releases certain hidden operating system resources that the stream needs in order to access the disk. After a stream is closed, it can't be written to.

Our code example will not compile, because lines 8, 10, and 11 throw exceptions. The constructor on line 8 throws `FileNotFoundException`. The `write()` call on line 10 and the `close()` call on line 11 throw `IOException`. So the code can be improved as follows:

```
1. import java.io.*;
2.
3. public class Write10Bytes
4. {
5.     public static void main(String[] args)
6.     {
7.         try
8.         {
9.             FileOutputStream fos;
10.            fos = new FileOutputStream("xyz");
11.            for (int i=0; i<10; i++)
12.                fos.write(i);
13.            fos.close();
14.        }
15.        catch (FileNotFoundException x)
16.        {
17.            System.out.println("Caught FileNotFoundException");
18.        }
19.        catch (IOException x)
20.        {
21.            System.out.println("Caught IOExn");
22.        }
23.    }
24. }
```

This code compiles, and it executes correctly. But it can be simplified a bit. `FileNotFoundException` is a subclass of `IOException`. So we can eliminate lines 13-16:

```
1. import java.io.*;
2.
3. public class Write10Bytes
4. {
5.     public static void main(String[] args_
6.     {
7.         try
8.         {
9.             FileOutputStream fos;
10.            fos = new FileOutputStream("xyz");
11.            for (int i=0; i<10; i++)
12.                fos.write(i);
13.            fos.close();
14.        }
15.        catch (IOException x)
16.        {
17.            System.out.println("Caught IOExn");
18.        }
19.    }
20. }
```

After this application runs, the current directory contains a 10-byte file named "xyz".

The Simple Output Lab animated illustration demonstrates an application that writes several bytes to a file. To run the program, type "`java io.SimpleOutputLab`". The initial display is shown in [Figure 13.1](#).



Figure 13.1: Simple Output Lab

The animation is very simple, but it will give you a good graphical image of the relationships between the data, the output stream, and the file. [Figure 13.2](#) shows the animation in progress.

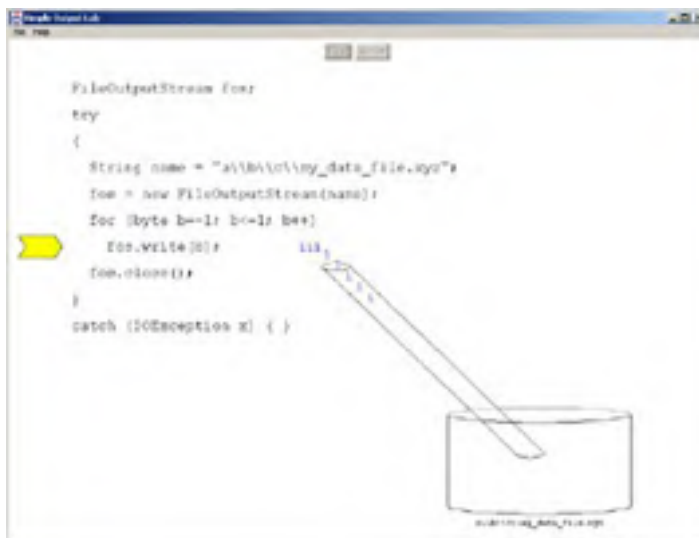


Figure 13.2: Simple Output Lab in progress

Reading Bytes

Reading bytes is almost exactly like writing bytes. You still have to do three things:

1. Construct an instance of `FileInputStream`.
2. Read the bytes.
3. Close the stream.

The following application reads back the file that was created in the [previous section](#):

```
1. import java.io.*;
2.
3. public class Read10Bytes
4. {
5.     public static void main(String[] args)
6.     {
7.         try
8.         {
9.             FileInputStream fis;
10.            fis = new FileInputStream ("xyz");
11.            for (int i=0; i<10; i++)
12.            {
13.                int theByte = fis.read();
14.                System.out.println(theByte);
15.            }
16.            fis.close();
17.        }
18.        catch (IOException x)
19.        {
```



```
20.         System.out.println("Caught IOExn");
21.     }
22. }
23. }
```

Line 8 creates the input stream, line 12 reads the bytes, and line 14 closes the stream. Line 12 prints out the bytes that were read, each one on its own line.

Note the strange variable on line 11. It is called `theByte`, but it is an `int`. The `read()` method of class `FileInputStream` reads a byte from the disk, but returns an `int`. Usually, the high-order 24 bits of the returned `int` are all 0s; the low-order 8 bits are the byte that was read from the disk. However, if the input stream has already read all the bytes in its file, the next `read()` call will return the `int` value -1. Recall that this value consists of 32 1's. This is distinct from the byte value of -1, which consists of eight 1's. If a file input stream reads such a byte from its file, the return value will have 1s in its low-order eight bits, and 0s in its high-order 24 bits. So there is no danger of confusing a byte read from the file whose value happens to be -1 with the `int` that signals that there is no more data in the file. [Table 13.1](#) makes this clear.

Table 13.1: Byte -1 vs. Int -1

byte -1, returned as an int	int -1, signaling end of file
00000000 00000000 00000000 11111111	11111111 11111111 11111111 11111111

You can use the special return value when you don't know the length of the file you are reading. In this example, suppose you don't know that the file contains 10 bytes. As you learned in [Chapter 5](#), when you don't know how many times the loop will execute, it's time to use a while loop:

```
import java.io.*;

public class Read10Bytes
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fis;
            fis = new FileInputStream ("xyz");
            while (true)
            {
                int theByte = fis.read();
                if (theByte == -1)
                    break;
                System.out.println(theByte);
            }
            fis.close();
        }
        catch (IOException x)
        {
            System.out.println("Caught IOException");
        }
    }
}
```

This application generates exactly the same output as the previous version, but this time there is no need to know the size of the file. This version can handle a file of any size.

The Simple Input Lab animated illustration demonstrates an application that reads several bytes from a file. To run the program, type "`java io.SimpleInputLab`". The animation is very simple, but like `SimpleOutputLab`, it will give you a good graphical image of the relationships between the data, the input stream, and the file. [Figure 13.3](#) shows the animation in progress.

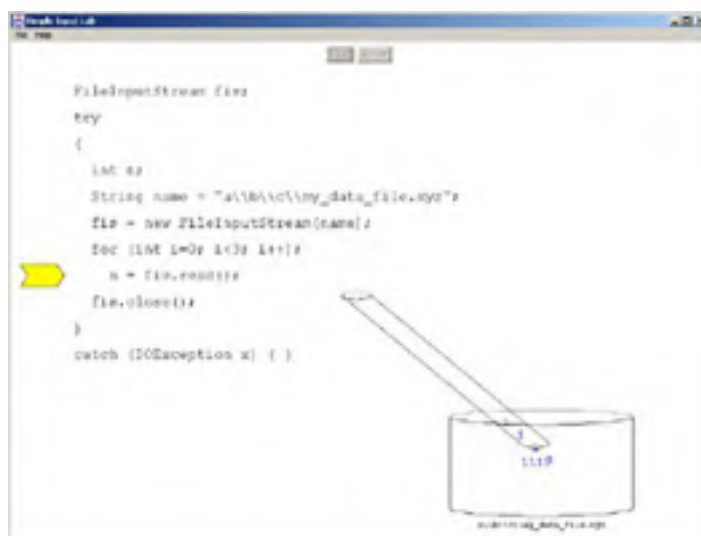


Figure 13.3: Simple Input Lab in progress

Team LIB

◀ PREVIOUS

NEXT ▶

Writing and Reading Data

At this point, you might be asking yourself, "When would I ever want to write or read bytes?" After all, one of the huge disadvantages of SimCom, as compared to the JVM, is that it deals only in bytes, while Java supports eight primitive data types and limitless class types.

The answer, fortunately, is that you never have to write or read bytes if you don't want to. You still have to create file input and output streams, and you still have to close them when you're done using them, but you don't have to write to them or read from them. Not directly, anyway. The writing and reading can be done by two very useful classes in the java.io package:

- `DataOutputStream`
- `DataInputStream`

The constructor for `DataOutputStream` takes a single argument. This argument is not the name of a file. Instead, it is a reference to a file output stream. Data written to a data output stream gets chopped up into bytes, which the data output stream passes to its file output stream. The technique of connecting streams together is called *chaining*. [Figure 13.4](#) shows a data output stream chained onto a file output stream.

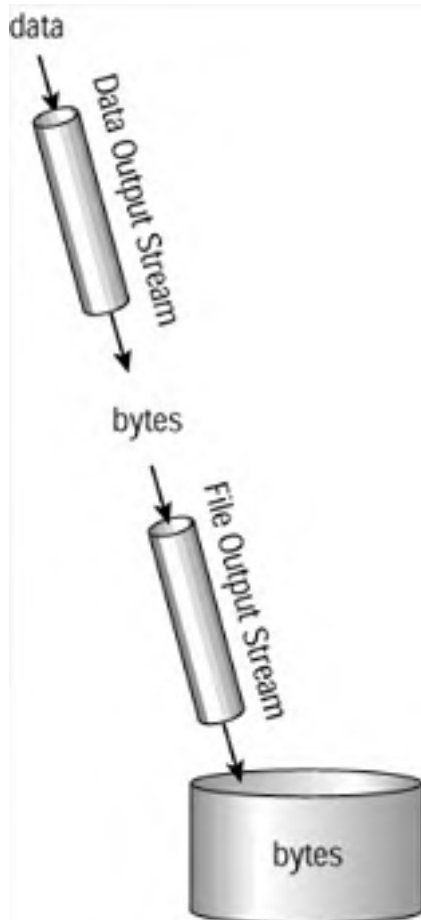


Figure 13.4: Output chaining

`DataOutputStream` has a large number of methods that chop up data and deliver bytes to the next stream in the chain. Here we will discuss nine of these methods:

- `writeBoolean(boolean boo)`
- `writeByte(int b)`
- `writeShort(int s)`
- `writeChar(int c)`
- `writeInt(int i)`
- `writeLong(long n)`
- `writeFloat(float f)`
- `writeDouble(double d)`

▪ writeUTF(String s)

It is obvious what the first eight methods do: They convert their primitive arguments into bytes. (It's surprising that `writeByte()`, `writeShort()`, and `writeChar()` take `int` args rather than the corresponding primitive types. That's just how it is.) What about UTF? Recall that Java's `char` type uses Unicode encoding. So a Java string is a run of Unicode characters. [UTF](#) is a standard for converting Unicode strings into bytes. Thanks to the `writeUTF()` method, you can use a data output stream to write any of Java's eight primitives, as well as any string. This is illustrated in the following application.

The following code chains a data output stream onto a file output stream, and then it writes one of each primitive type as well as one string:

```
import java.io.*;

public class WriteWithChain
{
    public static void main(String[] args)
    {
        boolean boo = true;
        byte b = 12;
        short sh = 12345;
        char c = 'M';
        int i = -654321;
        long n = 12341234;
        float f = 15;
        double d = 1.23e88;
        String s = "Where the devil did that dragon come from?";

        try
        {
            FileOutputStream fos;
            DataOutputStream dos;

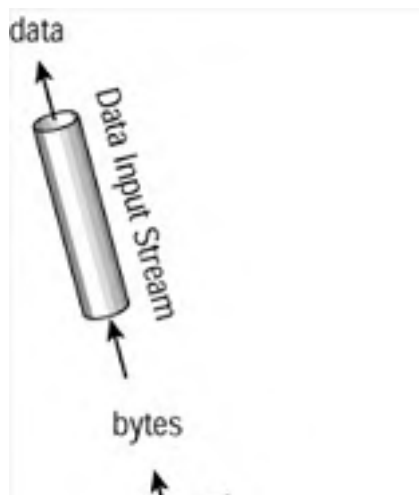
            fos = new FileOutputStream("abc");
            dos = new DataOutputStream(fos);
            dos.writeBoolean(boo);
            dos.writeByte(b);
            dos.writeShort(sh);
            dos.writeChar(c);
            dos.writeInt(i);
            dos.writeLong(n);
            dos.writeFloat(f);
            dos.writeDouble(d);
            dos.writeUTF(s);
            dos.close();
            fos.close();
        }

        catch (IOException x)
        {
            System.out.println("Caught IOException");
        }
    }
}
```

Note the two `close()` calls at the end of the try block. Every input and output stream should be closed after use. A good rule of thumb is to close chained streams in the opposite

order from their creation. Since the file output stream was constructed before the data output stream, close the data output stream first and the file output stream second.

Now you know how to write data to a file, so it is time to learn how to read data from a file. Again, you will chain a high-level stream onto a stream that communicates with a file. But this time, you will chain a data input stream onto a file input stream. [Figure 13.5](#) shows this arrangement.



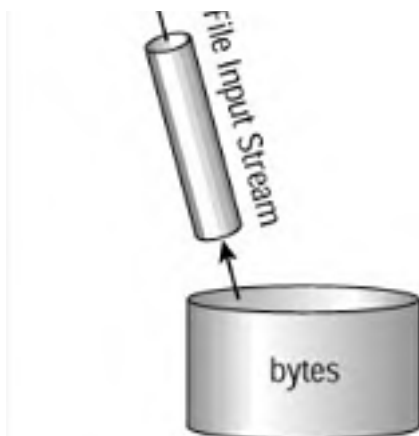


Figure 13.5: Input chaining

The `DataInputStream` class reads bytes from a lower-level stream, such as a file input stream has nine reading methods that correspond to the nine writing methods of `DataOutputStream`:

- `readBoolean()`
- `readByte()`
- `readShort()`
- `readChar()`
- `readInt()`
- `readLong()`
- `readFloat()`
- `readDouble()`
- `readUTF()`

These methods take no arguments. Their return types correspond to their names: `boolean` for `readBoolean()`, `byte` for `readByte()`, and so on. `readUTF()` returns a string. When any of these calls are made, the data input stream gets the appropriate number of bytes from its lower-level stream and assembles them to create the appropriate return value.

Now you can read the file by chaining a data input stream onto a file input stream:

```
import java.io.*;

public class ReadWithChain
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fis;
            DataInputStream dis;

            fis = new FileInputStream("abc");
            dis = new DataInputStream(fis);
            boolean boo = dis.readBoolean();
            System.out.println("Read boolean: " + boo);
            byte b = dis.readByte();
            System.out.println("Read byte: " + b);
            short sh = dis.readShort();
            System.out.println("Read short: " + sh);
            char c = dis.readChar();
            System.out.println("Read char: " + c);
            int i = dis.readInt();
            System.out.println("Read int: " + i);
            long n = dis.readLong();
            System.out.println("Read long: " + n);
            float f = dis.readFloat();
            System.out.println("Read float: " + f);
            double d = dis.readDouble();
            System.out.println("Read double: " + d);
            String s = dis.readUTF();
            System.out.println("Read string: " + s);
            dis.close();
            fis.close();
        }

        catch (IOException x)
        {
            System.out.println("Caught IOException");
        }
    }
}
```

```
}
```

This application's output is

```
Read boolean: true
Read byte: 12
Read short: 12345
Read char: M
Read int: -654321
Read long: 12341234
Read float: 15.0
Read double: 1.23E88
Read string: Where the devil did that dragon come from?
```

Obviously, this output reflects the data that was originally written to the file in the previous example. The two applications work together because the reading code reads exactly the same types, in exactly the same order, as were written by the writing code. Whenever you read from a file that was created with a data output stream, your read calls have to correspond exactly to the write calls that created the file. Otherwise, your data will be garbled beyond all recognition.

For example, suppose you mistakenly called `readLong()` instead of `readInt()`. The data input stream would grab the next eight bytes so that it could build a long. Those eight bytes would be the four-byte int (which is the next item of data in the file), and the first four bytes of the eight-byte long (which follows the int in the file).

The Data Chain Lab animated illustration demonstrates code that first writes three pieces of data to a file, and then reads them back. To run the application, type "`java io.DataChainLab`". [Figure 13.6](#) shows the display.



Figure 13.6: Data Chain Lab

In the three lines that write data, you will see pull-down choices for configuring which data type to write out. You can choose from any of the seven methods that write numerical types. You can also choose the values to be written. When you change the type being written, the corresponding reading code changes as well. This is in keeping with the rule that the type that is written must match the type that is read.

When you're ready, click the "Run" button to view the animation. If you want to run it again, perhaps with different output methods or values, first click "Reset". Then choose new methods and values, and click "Run" again.

[Figure 13.7](#) shows Data Chain Lab in progress. It has been configured to write and then read a byte, a long, and a double.

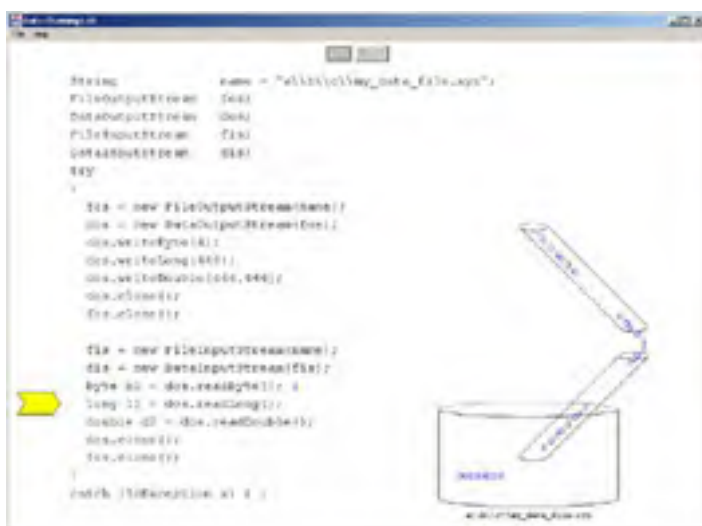


Figure 13.7: Data Chain Lab in progress: Text, writers, and readers

File output streams and file input streams are used for files that contain raw bytes. Data output streams and data input streams are used for files whose bytes represent multibyte data. There is a third kind of file, whose bytes represent text. For access to text files, Java provides classes called *readers* and *writers*.

Before presenting readers and writers, we should take a moment to explain what is meant by "text file". Recall that Java uses the modern Unicode scheme to represent text, using two bytes per character. This means there are 65,536 possible characters that can be represented. That's more than enough to represent every character of every language on the planet... until you consider Chinese, Japanese, and Korean. These non-phonetic alphabets have huge numbers of characters, enough to consume all 65,536 bit combinations. The international Unicode Consortium decides which characters of which languages will be represented by which bit combinations.

Well, that's the modern way to represent characters. It doesn't seem quite modern enough when you think about those Chinese, Japanese, and Korean symbols that get left out, but it's better than what we had before. The old way of doing things, from the invention of computers through the introduction of Unicode, was to use 8-bit characters. Every language group was on its own to decide which of the 256 possible bit combinations would represent which character. Most files created during that time used an encoding called *ASCII*, which stands for "American Standard Code for Information Interchange". ASCII encodes all the characters in American English, plus punctuation marks, into the range 0-127. The range 128-255 encodes symbols such as accented vowels, which are used in western European languages, as well as some Greek characters, line-drawing symbols, and some others. All of the characters that are represented in ASCII are represented in Unicode.

So here's the situation today: Within the JVM, characters are represented by Unicode. But in the world in general, there are millions of text files that use ASCII or other 8-bit representations. So Java needs a way to read those files and present their contents as Unicode strings. Also, Java needs a way to write ASCII files (as well as other 8-bit formats), because files can be read by non-Java programs that don't know about Unicode. Note that the problem cannot be solved by using data input and output streams that do lots of `readUTF()` and `writeUTF()` calls, because UTF is compressed Unicode, not ASCII.

Readers and writers solve the problem of translating between 16-bit characters within a JVM and 8-bit characters in text files. [Figure 13.8](#) illustrates the roles of readers and writers.

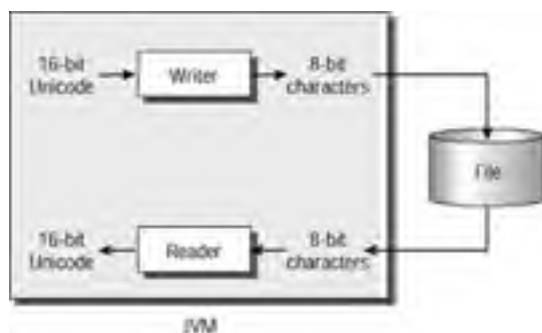


Figure 13.8: Readers and writers

`Reader` is an abstract class that reads 8-bit text and delivers Unicode chars. `Writer` is an abstract class that reads Unicode chars and delivers 8-bit text. For our purposes, the two most important subclasses of these two classes are `FileReader` and `FileWriter`, which read and write 8-bit text files.

A `FileWriter` is a lot like a `FileOutputStream`. You construct one, passing as an argument the name of the file you want to make. Then you write, and when you have finished, you close the `FileWriter`. This all must happen in a `try` block, because the code can throw `IOException` and some of its subclasses. The following code writes two lines of text to a file called `abc.txt`:

```
1. try
2. {
3.     FileWriter fw = new FileWriter("abc.txt");
4.     fw.write("Hello\n");
5.     fw.write("Goodbye\n");
6.     fw.close();
7. }
8. catch (IOException x) {}
```

Line 4 writes "Hello", followed by a newline character. Line 5 writes "Goodbye", followed by a newline character. Note that the newline is not automatic (as it is in the `System.out.println()` call, for example). If you want multiple lines of text, you have to indicate the line breaks yourself. So the following code creates an identical file:

```
1. try
2. {
3.     FileWriter fw = new FileWriter("abc.txt");
4.     fw.write("Hello\nGoodbye\n");
5.     fw.close();
6. }
7. catch (IOException x) {}
```

There are three common ways to indicate that a line has ended and a new line has begun:

- A return character (`'\r'`)
- A newline character (`'\n'`)
- A return character followed by a newline character

Which should you use? It depends on which other programs will be reading the file you create. Programs that run on Windows platforms expect a return character followed by a newline character. If you are creating files for Windows, you should do something like the following:

```
1. try
2. {
3.     FileWriter fw = new FileWriter("abc.txt");
4.     fw.write("Hello\r\nGoodbye\r\n");
5.     fw.close();
6. }
7. catch (IOException x) { }
```

If you run this code on a Windows machine and then double-click on the icon for the `abc.txt` file, Windows will open a Notepad window that displays (and lets you edit) the file. You can also open the file with any other program that reads text files, including Word.

You can read text files with the `FileReader` class, but this class is a bit limited. It is much easier to use the `LineNumberReader` class, where the `readLine()` method reads lines of text and returns strings. (This assumes that your text file has multiple lines, and that reading line by line will be useful to you. This is a safe assumption.) A call to `readLine()` reads one line from the input file. A line is a run of text, terminated by either a return character, a newline character, or a return character followed by a newline character. The line-termination characters are not part of the returned string.

A line number reader does not directly read from the input file. Rather, it is chained onto a file reader, in the same way a data input stream is chained onto a file input stream. [Figure 13.9](#) shows the relationship between a line number reader and a file reader.

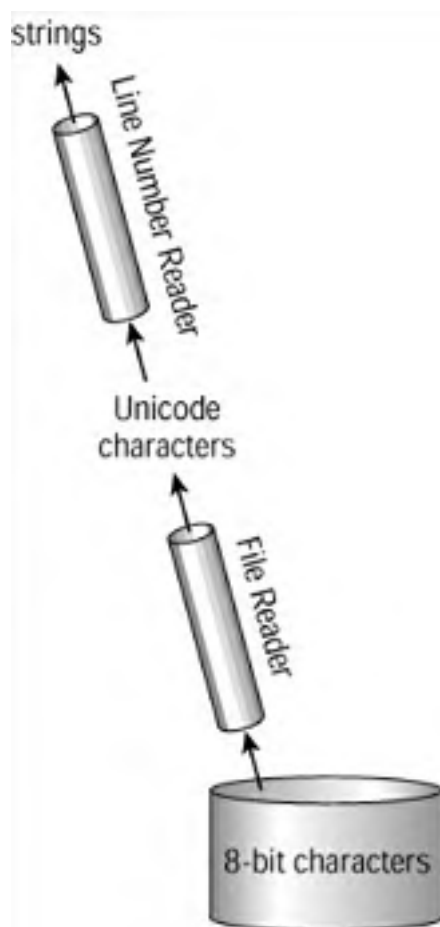


Figure 13.9: Line number reader and file reader

The following code reads and prints out the first two line of a character file:

```
try
{
    FileReader fr = new FileReader("zzz.txt");
    LineNumberReader lnr = new LineNumberReader(fr);
    System.out.println(lnr.readLine());
    System.out.println(lnr.readLine());
    lnr.close();
    fr.close();
}
catch (IOException x) { }
```

The `LineNumberReader` class keeps track of the number of lines it has read. You can retrieve the current line number by calling `getLineNumber()`.

The `readline()` method returns `null` if the end of the input file has been reached. So the following code prints out the line number of all lines in file `input.txt` that contain the word "purple":

```
1. try
2. {
3.     FileReader fr = new FileReader("input.txt");
4.     LineNumberReader lnr = new LineNumberReader(fr);
5.     String s = "";
6.     while (s != null)
7.     {
8.         s = lnr.readLine();
9.         if (s != null && s.indexOf("purple") != -1)
10.            System.out.println("Found \"purple\" at line " +
11.                               lnr.getLineNumber());
12.     }
13.     lnr.close();
14.     fr.close();
15. }
16. catch (IOException x) { }
```

The `while` loop runs as long as `s` is not `null` (that is, as long as the end of the file has not been reached). So `s` has to be initialized to anything besides `null` so that the loop will not immediately terminate. Line 9 calls `indexOf()` on the string returned by the line number reader. This method returns the position (in the string on which the method was called) of the string that is the method's argument. For example, if `s` in line 9 is "A ferocious purple dragon", the `indexOf()` call will return 12. If the argument string does not appear at all, `indexOf()` returns -1. So the condition in line 9 evaluates to `true` when the reader has not yet reached the end of the file, and the string just read contains "purple".

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. In this chapter, you learned about the following line:

```
String s = "C:my_backup\temporary\news";
```

What does the following code print out?

```
String s = "C:my_backup\temporary\news";  
System.out.println("***\n" + s + "***");
```

What is the moral of this exercise?

2. The code examples in the "[Writing and Reading Data](#)" section defined an int called `i`, a float called `f`, a double called `d`, and so on. But the long was called `n`, which breaks the pattern. You might have expected the long to be called `l`. Why do you think this was not done?
3. Write a program that creates a file containing 5,000 random doubles that are ≥ 0 and < 200 .
4. Write a program that verifies the file you created in the previous exercise. Your program should read the 5,000 doubles, making sure that each falls within the proper range. Your program should also make sure the file contains exactly 5,000 longs.
5. Look up the API documentation for the `java.io.File` class. An instance of this class contains information about an individual file. One of the methods of the class tells you the length in bytes of a file. Use this method to determine the number of bytes in the file you created in Exercise 3.

Chapter 14: Painting

We now begin a series of three chapters about visual programming. Up until now, all your applications have produced text output. In [Chapter 11, "Exceptions"](#), you learned how to provide text input via command-line arguments. Text input and output are fine, up to a point, but the mouse and the GUI provide a much richer environment for communicating a user's ideas to a program, and for communicating a program's results to a user.

Graphical user interface is usually abbreviated GUI. (Yes, it's pronounced "goeey.") The `java.awt` package contains dozens of classes that support GUI concepts like windows, colors, lines, squares, fonts, buttons, and check boxes. This chapter will show you how to display a window on your screen and paint basic shapes in it. That isn't spectacular, but this chapter will also prepare you for [Chapters 15, "Components,"](#) and 16, ["Events,"](#) where you will learn how to populate your GUIs with buttons, scrollbars, labels, and other standard controls. This chapter will end with an extended example program whose GUI combines custom painting with standard components.

Frames

A frame is a window on a computer screen, plus the "decoration" that makes it look like an independent window, plus the underlying programmatic behavior that lets you move windows around on your screen, resize them, iconify them, and so on. [Figure 14.1](#) shows a frame whose contents are gray.

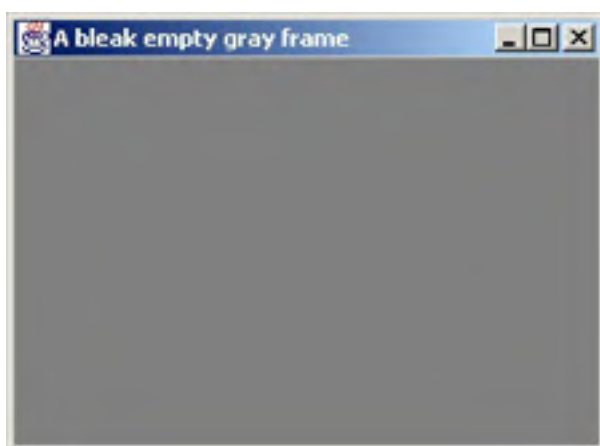


Figure 14.1: A frame with boring contents

The figure shows a Windows frame that was created by a Java program running on a Windows platform. Windows users will recognize the Minimize, Maximize, and Close buttons in the upper-right corner, as well as the decorations that give the outline its 3-D beveled appearance. On a different platform, the same program would create a frame whose controls and decorations looked slightly different, appropriate to the platform's windowing software.

So on any platform, a frame created by a Java program looks exactly like any other frame. This happens because the classes of the `java.awt` package do not directly draw components onto the screen. Instead, they instruct the underlying system's windowing software to do the work.

Note Java provides two alternative toolkits for creating GUIs. The simpler one is called AWT, which stands for Another Windowing Toolkit. The more complicated toolkit is called Swing, which doesn't stand for anything and is not discussed in this book. Swing does not use the underlying windowing software to draw components.

Here is the application that created the frame in [Figure 14.1](#):

```
1. import java.awt.*;
2.
3. public class EmptyFrame extends Frame
4. {
5.     EmptyFrame()
6.     {
7.         setTitle("A bleak empty gray frame");
8.         setBackground(new Color(128, 128, 128));
9.         setSize(300, 220);
10.    }
11.
12.    public static void main(String[] args)
13.    {
14.        EmptyFrame em = new EmptyFrame();
15.        em.setVisible(true);
16.    }
17. }
```

The application class extends `java.awt.Frame`. The superclass provides all the generic functionality of a frame. When you create a subclass of `java.awt.Frame`, you only have to provide the non-generic, application-specific behavior. In this example, the subclass does four things:

- Puts a message in the frame's title bar.

- Sets the frame's background color.
- Sets the frame's size.
- Makes the frame visible.

Line 7 sets the message in the title bar. The `setTitle()` method is inherited from the superclass; it takes a string argument.

Line 8 sets the frame's *background color*. Whenever the frame needs to be drawn on the screen, all of its pixels are set to the background color (except for the decoration pixels, of course). Then any application-specific drawing is performed. In this example, there is no application-specific drawing, so all you see in the frame's interior is uniform gray. The `setBackground()` method takes an argument of type `java.awt.Color`. We will look more deeply at this class in the [next section, "Colors."](#)

Line 9 uses the `setSize()` method to set the frame's size. The method's arguments are the desired width and height, in pixels. (A *pixel*, or *picture element*, is a dot on a display screen.) It's important to set a frame's size, because the default size is zero pixels wide by zero pixels high. So if you neglect to call `setSize()`, your frame will be too small to see.

Line 14 makes the frame visible. Before a frame executes `setVisible(true)`, it's just a lot of bytes somewhere in memory, like any other object. The first time `setVisible()` is executed, the frame establishes communication with the windowing software on the underlying system, and the windowing software draws the frame on the screen. The process is quite complicated, but it all happens automatically. You only need to remember to call both `setSize()` and `setVisible()`. Otherwise, your frame will not be seen.

Notice that the title bar message, foreground color, and size are set in the `EmptyFrame` constructor, while `setVisible()` is called in `main()`, after the frame has been constructed. The program would function the same if any of the calls on lines 7-9 were moved into `main()`. For example, the following code sets the background color and size in `main()`:

```
1. import java.awt.*;
2.
3. public class EmptyFrame extends Frame
4. {
5.     EmptyFrame()
6.     {
7.         setTitle("A bleak empty gray frame");
8.     }
9.
10.    public static void main(String[] args)
11.    {
12.        EmptyFrame em = new EmptyFrame();
13.        em.setBackground(new Color(128, 128, 128));
14.        em.setSize(300, 220);
15.        em.setVisible(true);
16.    }
17. }
```

This version produces an identical frame, but the previous version is considered better design. In the previous version, the constructor was responsible for setting the properties of the subclass instance, but it did not make the frame visible. This is clean design, because code that uses the class might want to create the object but not display it for a while. In general, constructors should set up the internal properties of an object without dictating when and how the object is to be used.

Note A frame that you create in Java does not automatically disappear when you click the "Close" button in its upper-right corner. The frame only sends an event to its listeners, using the mechanism that will be explained in [Chapter 16](#). To kill a frame, you can always type CONTROL-C in the console window where you started the program.

The code examples throughout the remainder of this book will feature frame subclasses whose constructors do everything except call `setVisible()`. Making the frame visible is the job of the code that uses the frame subclass. For us, this will always happen in `main()`, immediately after the subclass is created.

Now you know how to create a frame with boring uniform contents. Now it's time to learn how to put interesting things inside the frame. These things can be seen only if their colors are different from the frame's background color, so let's begin by looking at how Java handles colors.

Colors

Computer screens, like television screens, consist of rows and columns of tiny dots. It's difficult to see the dots with your unaided eye, but they are easy to see through a magnifying glass. The screen's electronics control each dot's color, under the direction of the computer's software or the TV's signal.

If you look closely at a pixel, you'll see that it consists of a red region, a green region, and a blue region. These are the pixel's *primary colors*.

This might contradict your childhood experiences. When you first started drawing or painting, you probably noticed that certain colors combine to make other colors. Mix red and blue to make purple. Red and yellow make orange, and blue and yellow make green. Other combinations aren't as pleasing: Red and green make an especially unpleasant brown. No doubt, someone explained to you that red, yellow, and blue are the primary colors that can be combined to make all other colors.

That isn't exactly true (there's no way to make white), but it pretty much explains the world of color. That is, until you stare too closely at a screen or start learning about computer colors. Then it seems that the primary colors are not red, yellow, and blue, but rather red, green, and blue.

Which trio is the real set of primary colors? It depends on your situation. When you mix paints, the pigments in the paints absorb certain colors from the ambient light. The remaining colors get reflected back into your eyes. Red, yellow, and blue (the primary colors of painting) are called *subtractive* primary colors because they are primary when light reflects off the absorbing pigment. Red, green, and blue (the primary colors of screens) are called *additive* primary colors because they are primary when different colors of light combine without pigment to absorb any hues. With additive primary colors, red plus green makes yellow, and red plus green plus blue makes white. You might have seen additive primaries in theaters or other venues that use colored spotlights. Where a red and green spotlight overlap, the light is yellow. Where red, green, and blue spotlights overlap, the light is white.

So when you control colors in a Java program, you have to think in terms of additive, not subtractive, primary colors. [Table 14.1](#) summarizes additive color mixing.

Table 14.1: Combining Additive Primary Colors

Primary Colors	Result
Red + green	Yellow
Red + blue	Magenta
Green + blue	Cyan
Red + green + blue	White

In Java, colors are represented by the `java.awt.Color` class. The constructor for this class has three arguments, which represent the amount of red, green, and blue that make up the color. The arguments range from a minimum of 0 through a maximum of 255. If all three arguments are 0, the color is black. If all three are 255, the color is white. As you can see from [Table 14.1](#), if red and blue are 255 while green is 0, the color is magenta. If red is 200, blue is 255, and green is 0, the color is a somewhat bluer magenta (because it contains less red).

The `Color` class has 13 predefined colors. These are public final static variables of type `Color`. (It may seem convoluted for a class to contain data of the same type as the class. That's just how it is.) The names of these variables are

- `Color.BLACK`
- `Color.WHITE`
- `Color.RED`
- `Color.GREEN`
- `Color.BLUE`
- `Color.YELLOW`
- `Color.CYAN`
- `Color.MAGENTA`
- `Color.ORANGE`
- `Color.PINK`
- `Color.LIGHT_GRAY`
- `Color.GRAY`
- `Color.DARK_GRAY`

If you want to use one of these colors, you don't have to create a new instance. For example, to set a frame's background color to orange, you can call `setBackground(Color.orange)`. If the 13 predefined colors don't give you what you want, you need to construct your own. You might find that `Color.orange` isn't intense enough. Its green level is 200, which tends to wash out the brilliance of the red. A nice intense orange is created by calling `new Color(255, 200, 0)`. You can make this the background color of a frame by calling `setBackground(new Color(255, 200, 0))`.

Looking at colors is better than reading about them. The Color Lab program lets you practice mixing primaries, and it also shows you all 13 predefined colors. To run the program, type `java visual.ColorLab`. [Figure 14.2](#) shows the initial display.

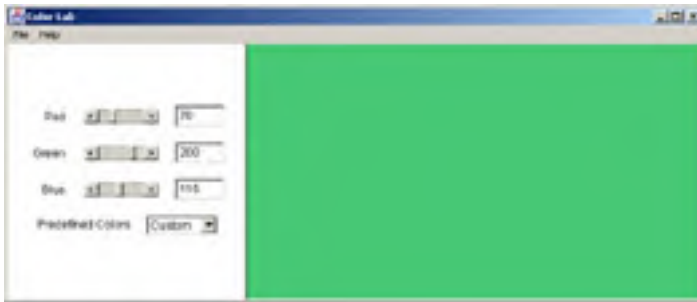


Figure 14.2: Color Lab

The control panel contains three scrollbars, with a text field showing the value of each scrollbar, as well as a pull-down menu. When you choose Custom from the pull-down menu, as shown in [Figure 14.2](#), the scrollbars are enabled. You can set them to any value from 0 through 255, and the display area to the right of the control panel will show the color you've specified. You can also type numbers into the text fields. Press Enter to make your entry take effect.

In addition to Custom, the pull-down menu lets you choose any of the 13 predefined colors of the `Color` class. When one of these is selected, the scrollbars and text fields are disabled. They display the red/green/blue levels of the selected color, but you can't use them for input. [Figure 14.3](#) shows Color Lab displaying a predefined color.



Figure 14.3: Color Lab with a predefined color

Notice that the sliders have no bubbles, and the numbers in the text fields are gray, indicating that those components are not enabled to receive user input.

Set the Color Lab inputs to display yellow. You can do this by selecting YELLOW, or by selecting Custom and manipulating the scrollbars. Now look at the yellow area of the screen through a magnifying glass. Observe the separate red and green areas of the individual pixels. Have a friend hold the magnifying glass steady, and move slowly backwards until the red and green seem to coalesce into yellow. How far from the screen are you when this begins to happen? You are invited to e-mail the distance to us at groundupjava@sgsware.com. We will compile the statistics and publish them on our website.

Now you know how to use Java's predefined colors, and how to construct a custom color when you need one. Now let's move on to using colors to draw shapes inside a frame.

Painting

There are many reasons why a frame's contents may need to be redrawn:

- The frame has become visible for the first time.
- The contents have changed due to user input (as when you moved a scrollbar in Color Lab).
- The frame has deiconified.
- The frame has moved to the front of the desktop, after having been covered or partially covered by another frame.

When any of these occur, the underlying windowing software notifies the frame, and a call is made to the frame's `paint()` method. The great thing about this arrangement is that you never have to detect these changes to the frame. The environment takes care of all that for you. All you have to do is subclass `Frame` and provide a `paint()` method that draws the frame's interior. In other words, you have to think about *what* to paint, but you don't have to think about *when* to paint.

When the environment decides that a frame needs painting, the frame's interior is cleared to its background color. By default, the background color is white. But as you saw in the [previous section](#), you can call `setBackground()` to set any background color you like. After that, the environment calls the frame's `paint()` method. The `paint()` version inherited from `Frame` does nothing at all. You are about to learn how to override `paint()` so that it does interesting things.

The argument of `paint()` is an instance of `java.awt.Graphics`. You might hear people call this object a *graphics context*, but it's more correct to call it a *graphics object*, and that is the name we will use. The graphics object is like an artist with a paintbrush, ready to paint the interior of a frame. It isn't a very talented artist (it only knows how to draw a few shapes), but it's very accurate. And as you'll see, it has excellent penmanship. You never have to construct an instance of `Graphics`; that's done for you by the environment. You just have to tell it what to paint.

An artist at work dips his brush in paint, brushes the paint onto paper, dips, brushes, and so on. The color that goes on the paper is, of course, the last color that the brush was dipped into. A graphics object works the same way. It has a method called `setColor()`, whose argument is a `Color`. It also has methods that draw shapes, including lines, rectangles, circles, and text messages. The shapes appear in the color that was the argument of the most recent `setColor()` call. So you can see that calling `setColor()` is like dipping your paintbrush into new paint. Another way to think of it is this: When you call `setColor()`, you set the color of all shapes to be drawn until the next `setColor()` call.

Now let's take a look at the different shapes that a graphics object can paint.

Drawing and Filling with a Graphics Object

The shapes that you can draw with the `Graphics` class include the following:

- Lines
- Squares and rectangles
- Circles and ovals

There are also methods that fill the interior of a square, rectangle, circle, or oval. All drawing happens in the color of the most recent `setColor()` method, as you saw in the [previous section](#). The methods have varying arguments that specify the size and location of the shape. All arrays are in units of pixels, not inches or millimeters. Horizontal positions are always called *x*, and are measured from the left edge of the frame. Vertical positions are called *y*, and are measured from the top of the frame. The location of a point is denoted by (*x*, *y*), as shown in [Figure 14.4](#).

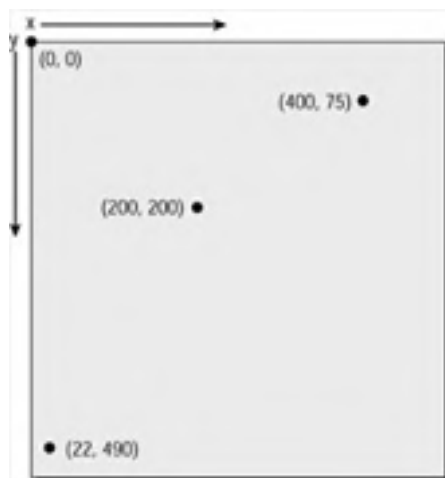


Figure 14.4: Pixel coordinates

The point at (0, 0) is called the *origin*. A frame's origin is its top-left pixel. Note that *x* increases from left to right, and *y* increases

from top to bottom. This is different from the Cartesian coordinates that you may have learned about in school, where y increases upward. The y-increases-downward scheme is standard in graphical programming, and it often causes confusion until people get used to it. It probably got its start in word-processing software, where line numbers increase from the top to the bottom of a document. Whatever its derivation might be, the scheme is here to stay.

To draw a line from (x0, y0) to (x1, y1), call the following on your graphics object:

```
drawLine(x0, y0, x1, y1);
```

The following code displays a frame with a black line on a white background:

```
1. import java.awt.*;
2.
3. public class BlackLineOnWhite extends Frame
4. {
5.     BlackLineOnWhite()
6.     {
7.         setSize(150, 180);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        g.setColor(Color.black);
13.        g.drawLine(60, 115, 120, 70);
14.    }
15.
16.    public static void main(String[] args)
17.    {
18.        BlackLineOnWhite blonw = new BlackLineOnWhite();
19.        blonw.setVisible(true);
20.    }
21. }
```

Figure 14.5 shows the frame.



Figure 14.5: A black line on a white background

The constructor just sets the frame's size. There's no need to set the background color explicitly, since you want the white default. Line 12 actually isn't required, because when `paint()` is called, the graphics object is set up to draw in black automatically.

To draw a rectangle, call the `drawRect()` method. Its four arguments are the x, y, width, and height of the rectangle, where (x, y) is the location of the rectangle's upper-left corner. The following code draws a blue rectangle that is 100 pixels wide by 35 pixels high, with its upper-left corner at (25, 50):

```
1. import java.awt.*;
2.
3. public class BlueRect extends Frame
4. {
5.     BlueRect ()
6.     {
7.         setSize(150, 180);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        g.setColor(Color.blue);
13.        g.drawRect(25, 50, 100, 35);
14.    }
15.
16.    public static void main(String[] args)
17.    {
18.        BlueRect br = new BlueRect();
19.        br.setVisible(true);
20.    }
21. }
```

Figure 14.6 shows the frame.



Figure 14.6: A rectangle

There is no separate method for drawing a square. You just call `drawRect()` with equal values for the width and height.

To draw an oval, you call `drawOval()` and specify the oval's *bounding box*. A bounding box is the smallest rectangle that encloses the oval. [Figure 14.7](#) shows several ovals and their bounding boxes.

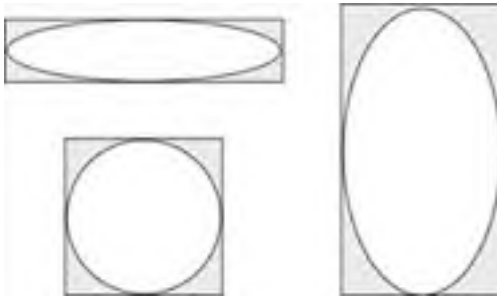


Figure 14.7: Ovals and bounding boxes

Notice that one of the shapes in [Figure 14.7](#) looks like a circle, not an oval. Actually, a circle is a kind of oval whose bounding box is a square.

The following code draws three ovals (shown in [Figure 14.8](#)), one of which is a circle:

```
1. import java.awt.*;
2.
3. public class ThreeOvals extends Frame
4. {
5.     ThreeOvals ()
6.     {
7.         setSize(150, 220);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        g.setColor(Color.black);
13.        g.drawOval(20, 40, 100, 35);
14.        g.drawOval(20, 85, 50, 60);
15.        g.drawOval(90, 105, 25, 25);
16.    }
17.
18.    public static void main(String[] args)
19.    {
20.        ThreeOvals throv = new ThreeOvals ();
21.        throv.setVisible(true);
22.    }
23. }
```





Figure 14.8: Three ovals

In the `drawOval()` calls, the arguments are the x, y, width, and height of the bounding box. Notice that in line 15, which draws the circle, the width and height are the same.

The `fillRect()` method draws a rectangle and fills its interior. The `fillOval()` method draws an oval and fills its interior. The following code displays two filled ovals and a filled rectangle:

```
1. import java.awt.*;
2.
3. public class Filled extends Frame
4. {
5.     Filled ()
6.     {
7.         setSize(200, 150);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        g.setColor(Color.black);
13.        g.fillOval(20, 50, 50, 20);
14.        g.fillRect(90, 40, 20, 70);
15.        g.fillOval(130, 50, 50, 20);
16.    }
17.
18.    public static void main(String[] args)
19.    {
20.        Filled f = new Filled();
21.        f.setVisible(true);
22.    }
23. }
```

The result is shown in [Figure 14.9](#).



Figure 14.9: Filled rectangle and ovals

Our last example in this section draws a filled oval that is centered in its frame. The oval is half as high and half as wide as the frame. The code uses the frame's `getSize()` method, which is inherited from one of the superclasses of `java.awt.Frame`. This method returns an instance of `Dimension`, which is a tiny class with two public ints called `width` and `height`:

```
1. import java.awt.*;
2.
3. public class CenteredOval extends Frame
4. {
5.     CenteredOval ()
6.     {
7.         setSize(200, 150);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        g.setColor(Color.black);
13.        Dimension size = getSize();
14.        g.fillOval(size.width/4, size.height/4,
15.                  size.width/2, size.height/2);
16.    }
17.
18.    public static void main(String[] args)
19.    {
20.        CenteredOval cenOv = new CenteredOval ();
21.        cenOv.setVisible(true);
22.    }
23. }
```

[Figure 14.10](#) shows the frame in its original size.



Figure 14.10: Original CenteredOval

If you replace line 7 with `setSize(400, 300);`, you get [Figure 14.11](#).

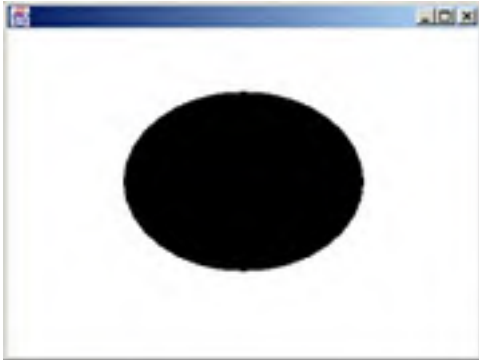


Figure 14.11: Resized CenteredOval

No matter what size is assigned to the frame in line 7, the `paint()` method always draws an oval with the correct proportions.

Now you know how to use a graphics object to do the following:

- Draw lines.
- Draw rectangles, including squares.
- Fill rectangles, including squares.
- Draw ovals, including circles.
- Fill ovals, including circles.

In the [next section](#), you'll learn how to draw text.

Text and Fonts

To draw text in a frame, call the graphics object's `drawString()` method. The method's arguments are the string to be drawn followed by its *x* and *y* position. The *x* position is the leftmost pixel of the first character in the string. The *y* position is the location of the baseline. The *baseline* of a string is the bottom of all characters except *g*, *j*, *p*, *q*, and *y*, which descend below the baseline. When you write on lined paper, the lines are baselines. [Figure 14.12](#) shows a string containing two characters that descend below the baseline.

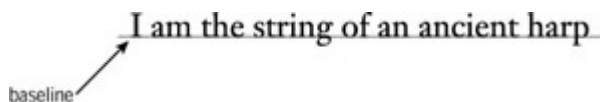


Figure 14.12: The baseline

The following code draws a string that contains six descending characters. It also draws the baseline in light gray:

```
1. import java.awt.*;
2.
3. public class FontAndBaseline extends Frame
4. {
5.     FontAndBaseline()
6.     {
7.         setSize(200, 150);
8.     }
9.
10.    public void paint(Graphics g)
11.    {
12.        int x = 25;
13.        int yBaseline = 100;
14.        g.setColor(Color.lightGray);
15.        g.drawLine(x, yBaseline, 125, yBaseline);
16.        g.setColor(Color.black);
17.        g.drawString("just a gaping quay", x, yBaseline);
18.    }
19.
20.
21.    public static void main(String[] args)
22.    {
23.        FontAndBaseline fabl = new FontAndBaseline();
24.        fabl.setVisible(true);
25.    }
26. }
```

The `drawString()` call is on line 17. [Figure 14.13](#) shows the frame.



Figure 14.13: Text and baseline in a frame

You can use the graphics object's `setFont()` method to control the font in which text is displayed. Calling `setFont()` before calling `drawString()` is a bit like calling `setColor()` before drawing or filling a shape. The `setFont()` call determines the font of all subsequent `drawString()` calls, until the next `setFont()` call.

A font has three properties:

- Style
- Size
- Family

The style can be either plain, **bold**, *italic*, or **bold-italic**. The size is in pixel units.

The font families that are available to you vary from one machine to the next, but there are three that you can always count on:

- Monospaced
- Serif
- Sans Serif

In Monospaced font, all characters are spaced equally. The spacing is determined by the font's size. It's difficult to read a dense block of text in Monospaced font, but it's ideal for source code. All the code listings in this book appear in Monospaced font.

Serif is a *variable-width font*, which means that characters have different widths. For example, *i* is narrower than *m*. This book is printed in a variable-width font. Notice how *iiiiiiii* is much narrower than *mmmmmmmm*, even though both are 10 letters long. Variable-width fonts are designed for easy reading of dense text, such as you see in a book or newspaper. This font uses *serifs*, which are small decorations on the tips of letters. Serifs improve readability in medium to large fonts, but are annoying in small fonts.

Sans Serif is a variable-width font that does not use serifs. ("Sans" is French for "without".) This font works best when the font is small enough that serifs would interfere with readability.

To use a font in Java, you must first construct an instance of the `java.awt.Font` class. The constructor takes three arguments: the family, the style, and the size. The family is a string; the style and size are ints. For these three families, the strings are `Monospaced`, `Serif`, and `SansSerif`. For plain, bold, and italic, the `Font` class provides public final static ints named `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. For bold-italic style, use `Font.BOLD + Font.ITALIC`.

After you construct an instance of `Font`, you can pass it into the `setFont()` method of a graphics object. The following code sets a 36-point bold-italic Serif font:

```
Font f = new Font("Serif", Font.PLAIN+Font.BOLD, 36);  
g.setFont(f);
```

When these lines are inserted between lines 16 and 17 in the previous example (that is, just before the `drawString()` call), the result is as shown in [Figure 14.14](#).



Figure 14.14: Text in a frame

Your computer probably has dozens of fonts in addition to the three standard ones. Many of them may be more interesting and playful than Monospaced, Serif, and Sans Serif. The more stylized fonts tend to be appropriate in more limited situations.

When you use a non-standard font in a Java application, be aware that you're taking a risk. It's possible that the font won't be available on all computers that will be running your application. When this happens, all characters will appear on the screen as small empty rectangles.

The `GraphicsEnvironment` class contains information about a computer's graphics system, including the names of all available fonts. The class has a static method called `getLocalGraphicsEnvironment()`, which returns an instance of the class with all the data fields set to reflect the capabilities of the underlying computer. Another call is `getAvailableFontFamilyNames()`, which is not static. It returns the font families as an array of strings. So, to retrieve the array, you can do something like the following:

```
GraphicsEnvironment grenv =  
    GraphicsEnvironment.getLocalGraphicsEnvironment();  
String[] names = grenv.getAvailableFontFamilyNames();
```

The `Font Lab` program lets you see all the fonts that are available on your computer. To run it, type `java visual.FontLab`. You will see the GUI shown in [Figure 14.15](#).



Figure 14.15: Font Lab

The pull-down menu in `Font Lab`'s control panel lets you choose from among all of the fonts on your machine, as detected by `getAvailableFontFamilyNames()`. You can change the family, style, and size. Some families do not support all styles. Some have only plain and italic, and others have only plain. [Figure 14.16](#) shows one of the more exotic fonts.

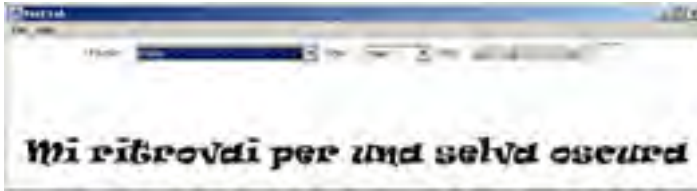


Figure 14.16: Font Lab with an exotic font

Play around with the various fonts. Find one that seems serious and one that seems playful. How do their shapes differ? Select 16-point plain Serif. Reduce the size one point at a time until the font becomes hard to read. Do the same for SansSerif. You will probably find that SansSerif remains readable down to a slightly smaller size than Serif.

Team LiB

← PREVIOUS NEXT →

Frame Lab

The Frame Lab animated illustration lets you practice setting colors, drawing shapes, setting fonts, and drawing text. To run the program, type `java visual.FrameLab`. The initial display is shown in [Figure 14.17](#).



Figure 14.17: Initial Frame Lab display

The animated illustration simulates a subclass of `Frame`, with a `paint()` method that you can configure. You can enter any class name you like in the first text field. (Notice that the lines that declare and call the constructor change as you change the name.) If you want your constructor to set the background color, make sure the check box in the `setBackground` line of the constructor is checked, and select a color from the pull-down menu. The constructor sets the frame's size to 450 by 450, but you can enter any number you want.

The body of the `paint()` method consists of ten lines, each controlled by its own pull-down menu that lets you select a method to call on the graphics object. You can set the color or the font, or you can call any of the drawing methods that were presented in this chapter. You can also select a comment (`/******`), which indicates that you don't want the line to do anything. No matter what you select, the line will present you with controls for entering the arguments of the method you've chosen.

When you're ready to simulate execution of the code you've set up, click either `Run` or `Run Lightspeed` to view either an animation or an instant result. If you want to run again, click `Reset`, adjust the controls, and again click either `Run` or `Run Lightspeed`. [Figure 14.18](#) shows one possible Frame Lab configuration.



Figure 14.18: Frame Lab with custom configuration

The resulting frame is shown in [Figure 14.19](#).



Figure 14.19: The result of [Figure 14.18](#)

Try configuring Frame Lab to paint the following:

- A line of text, in any font you like, with the baseline visible.
- A line of text centered in a filled oval.
- Three concentric circles.

Now draw anything you like. If you create any interesting results that you would like to share, please email them to us at groundupjava@sgsware.com. In the next edition of this book, Frame Lab will include a gallery of the best pictures submitted, along with the artists' names.

Take a look at the `main()` method in Frame Lab. So far in this chapter, all your `main()` methods have been two lines long, like this:

```
public static void main(String[] args)
{
    FontAndBaseline fabl = new FontAndBaseline();
    fabl.setVisible(true);
}
```

In Frame Lab, vertical space is a valuable commodity. That's why the open curly brackets appear at the ends of lines, rather than on their own lines. Frame Lab's `main()` is only one line long:

```
(new FancyFrame()).setVisible(true);
```

This is just a shorter equivalent of the following:

```
FancyFrame ff = new FancyFrame();
ff.setVisible(true);
```

In the single-line version, there is no reference to the instance of `FancyFrame` that gets constructed. If you want to make another call on the instance after `setVisible()`, you're out of luck. The instance is *anonymous*. "Anonymous" means "without a name," and a reference is like an object's name. The single-line version of the constructor is considered better style, because there is no need for the reference except in the `setVisible()` call. For the rest of this book, the `Frame` subclass instances will be anonymous.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. The first code example in this chapter used the following code to set a frame's background color:

```
setBackground(new Color(128, 128, 128));
```

Describe the color that this line creates.

2. Run Color Lab, and adjust the scrollbars so that the displayed color matches something you can see (a piece of clothing you're wearing, or something on your desk, or anything else you like). Now write an application that displays a frame whose interior is the color you've chosen.
3. One of the code examples in this chapter used the `getSize()` method, which `Frame` inherits from one of its superclasses. Use the API to find out which superclass implements the method.
4. Write a program that draws a five-pointed star. Your frame should be 400 x 400 pixels. The coordinates of the star's points are (200, 375), (97, 58), (366, 254), (34, 254), and (303, 58). The easy way is to write a `paint()` method that calls `drawLine()` five times. But that approach isn't ideal, because you have to type each `x` and each `y` twice. (Each point is the end of two lines, so it appears in two `drawLine()` calls.) Typing data, code, or anything else more than once is considered bad style. If one of the copies has a typo and doesn't match the original precisely, your program won't function correctly. To avoid duplication of data, your program should have two `int` arrays, defined as follows:

```
int[] xs = {200, 97, 366, 34, 303};  
int[] ys = {375, 58, 254, 254, 58};
```

Your `paint()` method should have a loop that accesses these arrays. `drawLine(...)` should appear only once in your code, inside the loop.

5. Write a program that lists all the font families that are available on your computer.

Chapter 15: Components

A *component* is a GUI device that presents user input to programs and displays program information to users. Standard GUI components include buttons, text fields, scrollbars, and menus. Java's `awt` package provides a rich suite of components, all of which are reasonably easy to use. A program's GUI can combine custom drawing, which you learned about in the [previous chapter](#), with standard components, which you will learn about here and in the [next chapter](#).

You have probably heard the expression "[look and feel](#)." For example, several years ago there was a major lawsuit between Apple and Microsoft; one company alleged that the other had plagiarized their look and feel. A program's *look and feel* consists of its appearance (look) plus its responses to user input (feel). This chapter will focus on look; feel will be covered in [Chapter 16](#), "[Events](#)."

A Survey of Components

In this section, you will learn about some of the most useful components of the `awt` package:

- Buttons
- Checkboxes
- Choices
- Labels
- Menus
- Text fields
- Text areas
- Scrollbars

You have probably encountered all of these component types in the course of using your computer. As a reminder, [Figure 15.1](#) shows one of each component type listed.



Figure 15.1: A component sampler

Now let's jump in and learn about each of these components.

Buttons

Buttons are perhaps the most familiar of all component types. We are so accustomed to them that we take for granted statements like, "Click on the OK button on your screen to confirm your purchase." Of course, there isn't really a button on the screen; it's just a picture of a button. "Press the OK button" really means, "Move your mouse until the arrow on the screen is over the picture of the button. Then press and release the button on your mouse."

[Figure 15.2](#) shows a button in a frame.





Figure 15.2: A button in a frame

In Java, buttons are represented by the `java.awt.Button` class. Here is the code that created [Figure 15.2](#):

```
1. import java.awt.*;
2.
3. class ShowButton
4. {
5.     public static void main(String[] args)
6.     {
7.         Frame f = new Frame("Simple Button");
8.         LayoutManager lom = new FlowLayout();
9.         f.setLayout(lom);
10.        Button btn = new Button("Hello");
11.        f.add(btn);
12.        f.setSize(200, 200);
13.        f.setVisible(true);
14.    }
15. }
```

Line 10 illustrates the most common `Button` constructor, which takes a string argument. The string appears as the button's text label.

Something in this code is conspicuously absent, and something else in conspicuously mysterious. Look at [Figure 15.2](#). The button is a reasonable size. It's big enough to encompass its text, and not much bigger. It's near the top of the frame, and it's horizontally centered. Conspicuously absent is the code that sets the button's size and location.

The mysterious code is lines 8 and 9, which construct and use an instance of `FlowLayout`. The `FlowLayout` class is a kind of *layout manager*. Layout managers are responsible for setting the location and size of components. We will visit them in detail in the second half of this chapter; you will find them much easier to understand after you know about components. For now, be aware that the button's reasonable size and location were set by the layout manager. The call to `add()` on line 11 puts the button in the frame. The method uses the layout manager to work out the details.

The example code is not very object-oriented. Here is a version that extends `Frame`:

```
1. import java.awt.*;
2.
3. class BtnInAFrame extends Frame
4. {
5.     public BtnInAFrame()
6.     {
7.         setLayout(new FlowLayout());
8.         Button btn = new Button("Hello");
9.         add(btn);
10.        setSize(200, 200);
11.    }
12.
13.    public static void main(String[] args)
14.    {
15.        (new BtnInAFrame()).setVisible(true);
16.    }
17. }
```

The GUI that this code produces is identical to the previous example. Notice that construction and use of the layout manager have now been combined into a single line (line 7) so as to be less obtrusive.

In the [previous chapter](#), you learned about fonts and colors. The `Button` class has three methods that let you control the font and color of a button:

- `setFont(Font f)`
- `setForeground(Color c)`
- `setBackground(Color c)`

Actually, `Button` inherits these methods from its superclass, `java.awt.Component`. All the component classes you will learn about in this chapter extend `java.awt.Component`, so they all implement these three methods.

`setFont()` sets the font of any text that the component displays. `setForeground()` sets the color of the component's text, and `setBackground()` sets the component's background color. The following code displays a button with a large yellow serif font on a blue background:

```
import java.awt.*;

class FancyButtonInFrame extends Frame
{
    public FancyButtonInFrame()
    {
        LayoutManager lom = new FlowLayout();
        setLayout(lom);
        Button btn = new Button("Hello");
        Font font = new Font("Serif", Font.ITALIC, 36);
        btn.setFont(font);
        btn.setForeground(Color.yellow);
        btn.setBackground(Color.blue);
        add(btn);
        setSize(200, 200);
    }
}
```

```
}  
  
public static void main(String[] args)  
{  
    FancyButtonInFrame b = new FancyButtonInFrame();  
    b.setVisible(true);  
}  
}
```

Figure 15.3 shows the button in the frame. The black-and-white screen shot does not do justice to the colors, but you can clearly see the enlarged italic font.



Figure 15.3: A fancy button

Notice that the button is still large enough to encompass its text, even though the text is now considerably larger. We have the mysterious layout manager to thank for that.

Buttons are for clicking. If you run any of the code in this section, you will find that the buttons do the right thing when you click on them: They look like they're indented into the screen until you release the main mouse button. However, nothing happens within the program. This should be expected, since there is no code in any of the programs that appears to deal with listening for button input. Listening for input is a function of feel, not look, so it will be presented in the [next chapter](#).

Checkboxes

A checkbox is a little box that can be either checked or not checked. The checked/not checked state changes whenever the user clicks on the component. The two most useful constructors are

- `Checkbox(String s)`
- `Checkbox(String s, boolean state)`

The string is the checkbox's text. The boolean in the second form is the checkbox's initial state. The following code builds and displays a simple unchecked checkbox:

```
import java.awt.*;  
  
class CboxInnaFrame extends Frame  
{  
    public CboxInnaFrame ()  
    {  
        setLayout(new FlowLayout());  
        Checkbox cbox = new Checkbox("Check Me");  
        add(cbox);  
        setSize(200, 200);  
    }  
  
    public static void main(String[] args)  
    {  
        (new CboxInnaFrame ()).setVisible(true);  
    }  
}
```

The result is shown in [Figure 15.4](#).





Figure 15.4: A simple checkbox

The figure just shows the initial state of the GUI. If someone clicks on the box, it will be checked.

The following code displays a checkbox that is checked if the application was invoked with "yes" as its first command-line argument.

```
import java.awt.*;

class CheckedCbox extends Frame
{
    public CheckedCbox(boolean b)
    {
        setLayout(new FlowLayout());
        Checkbox cbox = new Checkbox("Check Me", b);
        add(cbox);
        setSize(200, 200);
    }

    public static void main(String[] args)
    {
        boolean state = false;
        if (args.length > 0 && args[0].equals("yes"))
            state = true;
        (new CheckedCbox(state)).setVisible(true);
    }
}
```

Figure 15.5 shows the result when the application is invoked by typing `java CheckedCbox yes`.



Figure 15.5: A checked checkbox

Now it's time to display several components together. The following code creates three checkboxes and a button:

```
1. import java.awt.*;
2.
3. class Boats extends Frame
4. {
5.     Checkbox[] cboxes;
6.     Button btn;
7.     String[] sizes = { "small", "medum", "large" };
8.
9.     Boats()
10.    {
11.        setLayout(new FlowLayout());
12.
13.        cboxes = new Checkbox[sizes.length];
14.        for (int i=0; i<sizes.length; i++)
15.        {
16.            String s = "a " + sizes[i] + " boat";
17.            cboxes[i] = new Checkbox(s);
18.            add(cboxes[i]);
19.        }
20.        btn = new Button("Add to shopping cart");
21.        add(btn);
22.
23.        setSize(600, 200);
24.    }
25.
26.    public static void main(String[] args)
27.    {
28.        new Boats().setVisible(true);
29.    }
30. }
```

This application is slightly longer than any of our previous GUI code examples. It is long enough to warrant some structure. Note that the three checkboxes, which could have been constructed one by one, are constructed in a loop. Line 16 generates the text for each checkbox, based on the appropriate string from the `sizes` array on line 7. A nice benefit of this structure is the visual isolation of the literal strings. They are easy to find, up near the top of the code listing.

Did you notice that "medium" was misspelled? Would you have noticed so easily if the literal strings were in the middle of the code? Imagine the difficulty in correcting a spelling error if the strings were scattered over a 300-line constructor. When you misspell a keyword (like "for" or "new"), the compiler tells you the line number where the error occurs. But when you misspell a literal string, the compiler can't help you. You have to go hunting for the string.

Another benefit of this program's structure is the ease with which it can be modified. If you want to add or delete some sizes, changing the array on line 7 is the only change you need to make. Exercises 2 and 3 show this in action.

Figure 15.6 shows the example program's GUI. (The spelling error has been fixed.)



Figure 15.6: Three checkboxes and a button

In the figure, the user has checked both "a small boat" and "a large boat". This is suspicious. A GUI should capture the user's precise intention. Moreover, a well-designed GUI should make it impossible for a user to enter invalid data. Figure 15.6 gives the impression that the user is supposed to check only one of the three checkboxes. If the checkboxes represent mutually exclusive alternatives, the GUI should be changed to discourage (or, better yet, prevent) selection of more than one boat size.

There are two ways to change the GUI:

- Insert text that tells the user to check only one box.
- Insert code that automatically unchecks a box whenever the user makes a new selection.

The first option puts all the responsibility on the user. The GUI still permits invalid input, and the user gets all the blame when something goes wrong. This approach is unforgivable. It's also distressingly common: Every Web user has experienced an extreme version of it. Think of the last time you typed your credit card number or phone number into a Web page, only to be told that you should have (or should not have) used spaces or hyphens. Then you have to wait for the page to reload, you have to reenter your credit card or phone number, and if the page designer was especially inept, you have to reenter your name and address as well.

The second option makes it impossible for any user to select more than one option. The result is a GUI that is free from blame. This is the approach we will take.

Java's checkboxes can act as radio buttons. A *radio button* is a member of a group, only one of which can be selected at any time. The term comes from the station-selection buttons on a car radio. To give radio-button behavior to a group of checkboxes, you first create an instance of the class `java.awt.CheckboxGroup`:

```
CheckboxGroup cbg = new CheckboxGroup();
```

When you construct your checkboxes, use one of the following constructors:

```
Checkbox(String s, boolean state,  
         CheckboxGroup cbg)
```

or

```
Checkbox(String s, CheckboxGroup cbg,  
         boolean state)
```

Here is the previous example, rewritten to use a checkbox group:

```
1. import java.awt.*;  
2.  
3. class RadioBoats extends Frame  
4. {  
5.     Checkbox[]    cboxes;  
6.     Button        btn;  
7.     String[]     sizes = { "small", "medium", "large" };  
8.  
9.     RadioBoats()  
10.    {  
11.        setLayout(new FlowLayout());  
12.  
13.        cboxes = new Checkbox[sizes.length];  
14.        CheckboxGroup cbg = new CheckboxGroup();  
15.        for (int i=0; i<sizes.length; i++)  
16.        {  
17.            String s = "a " + sizes[i] + " boat";  
18.            boolean state = (i == 0);  
19.            cboxes[i] = new Checkbox(s, state, cbg);  
20.            add(cboxes[i]);  
21.        }  
22.        btn = new Button("Add to shopping cart");
```

```
22.     btn = new Button( "Add to shopping cart" );
23.     add(btn);
24.
25.     setSize(600, 200);
26. }
27.
28. public static void main(String[] args)
29. {
30.     new RadioBoats().setVisible(true);
31. }
32. }
```

Line 18 creates a boolean whose value is `true` in the first pass through the loop. Thus, the first checkbox is checked and the rest are not. [Figure 15.7](#) shows the GUI after the "a large boat" box has been selected.



Figure 15.7: Checkboxes as radio buttons

Notice the appearance of the checkboxes. There are no check marks, and there are no boxes. The circular buttons are a standard visual cue that the components have radio behavior. This cue is standard not just in Java, but in all current windowing toolkits.

When you create a GUI that has multiple checkboxes, ask yourself if the checkboxes can be selected independently, or if only one should be selected at any moment. If they are independent, use plain checkboxes. If they are exclusive, give them radio behavior by creating a checkbox group for them.

Choices

Suppose you want to create a GUI for specifying a font. Suppose also that you want your users to choose one of the three standard font families, and also to choose a size from among a small set of options. You might use the following code:

```
import java.awt.*;

class ChooseFontByRadios extends Frame
{
    String[] families = {"Monospaced", "Serif", "SansSerif"};
    int[] sizes = {16, 24, 32, 64};

    public ChooseFontByRadios()
    {
        setLayout(new FlowLayout());

        CheckboxGroup familyCBG = new CheckboxGroup();
        for (int i=0; i<families.length; i++)
            add (new Checkbox(families[i], (i==0), familyCBG));

        CheckboxGroup sizeCBG = new CheckboxGroup();
        for (int i=0; i<sizes.length; i++)
            add (new Checkbox(""+sizes[i], (i==0), sizeCBG));

        setSize(500, 200);
    }

    public static void main(String[] args)
    {
        (new ChooseFontByRadios()).setVisible(true);
    }
}
```

This code creates two checkbox groups. The result, as you can see in [Figure 15.8](#), is less than brilliant.



Figure 15.8: Multiple checkbox groups

The problem with the GUI is that there are no visual cues to tell you that there are two independent groups of checkboxes. This brings us to an important principle of GUI design: Components that are *functionally* related should also be *visually* related. To create a sensation of visual relationship among a group of components, you need to do two things:

- Place the components near one another.
- Isolate them from other nearby components.

In [Figure 5.8](#), the components that control the font family are certainly near one another, but they are not isolated from the components that control size.

The `java.awt.Choice` component class offers an alternative to groups of checkboxes. A *choice* is a single component that lets the user make a one-of-many selection. [Figure 15.9](#) shows a simple choice.



Figure 15.9: A choice

Choices are like pull-down menus. When you click on the component, the entire set of options is displayed, as shown in [Figure 15.10](#).



Figure 15.10: An expanded choice

Here is the code that created the GUIs in [Figures 15.9](#) and 15.10:

```
1. import java.awt.*;
2.
3. class SimpleChoice extends Frame
4. {
5.     String[] families = {"Monospaced", "Serif",
6.                         "SansSerif"};
7.
8.     public SimpleChoice()
9.     {
10.        setLayout(new FlowLayout());
11.        Choice c = new Choice();
12.        for (int i=0; i<families.length; i++)
13.            c.add(families[i]);
14.        add(c);
15.        setSize(200, 200);
16.    }
17.
18.    public static void main(String[] args)
19.    {
20.        (new SimpleChoice()).setVisible(true);
21.    }
22. }
```

The choice is created in line 11. Notice that there are no constructor arguments. Line 13 calls the choice's `add()` method to add more options; the method's argument is a string. After the choice is constructed and populated, it is added to the GUI at line 14. Notice that line 14 uses the `add()` method of the frame to add the choice component to the frame. This is a different method from the `add()` in line 13, which adds options to the choice.

The following application uses choices to support selection of a font family and size:


```
1. import java.awt.*;
2.
3. class FontChoice extends Frame
4. {
5.
6.     String[]  families = {"Monospaced", "Serif",
7.                          "SansSerif"};
8.     int[]     sizes    = {16, 24, 32, 64};
9.
10.    public FontChoice()
11.    {
12.        setLayout(new FlowLayout());
13.        Choice c = new Choice();
14.        for (int i=0; i<families.length; i++)
15.            c.add(families[i]);
16.        add(c);
17.        c = new Choice();
18.        for (int i=0; i<sizes.length; i++)
19.            c.add(""+sizes[i]);
20.        add(c);
21.        setSize(200, 200);
22.    }
23.
24.    public static void main(String[] args)
25.    {
26.        (new FontChoice()).setVisible(true);
27.    }
28. }
```

The resulting GUI is shown in [Figure 15.11](#).



Figure 15.11: Two choices

You can see that the choice component does an excellent job of isolating its parts visually.

Labels

Labels are by far the simplest Java components. They are the only components that cannot be used to gather user input. They just sit there. Often labels appear next to scrollbars, text fields, choices, or other components that do not have their own labels.

A label looks like text on a screen, exactly as if it had been painted there by the `drawString()` method of the `Graphics` class, which you saw in the [previous chapter](#). The following code adds labels to the GUI of the last example in the [previous section](#):

```
1. import java.awt.*;
2.
3. class FontChoiceWithLabels extends Frame
4. {
5.     String[]  families = {"Monospaced", "Serif",
6.                          "SansSerif"};
7.     int[]     sizes    = { 16, 24, 32, 64 };
8.
9.     public FontChoiceWithLabels()
10.    {
11.        setLayout(new FlowLayout());
12.        Label familyLabel = new Label("Font family:");
13.        add(familyLabel);
14.        Choice c = new Choice();
15.        for (int i=0; i<families.length; i++)
16.            c.add(families[i]);
17.        add(c);
18.        Label sizeLabel = new Label("Font size:");
19.        add(sizeLabel);
20.    }
21. }
```

```
20.     c = new Choice();
21.     for (int i=0; i<sizes.length; i++)
22.         c.add(""+sizes[i]);
23.     add(c);
24.     setSize(350, 200);
25. }
26.
27. public static void main(String[] args)
28. {
29.     (new FontChoiceWithLabels()).setVisible(true);
30. }
31. }
```

Lines 12 and 18 construct labels. The constructor takes a single argument, which is the label's text. [Figure 15.12](#) shows this code's GUI.

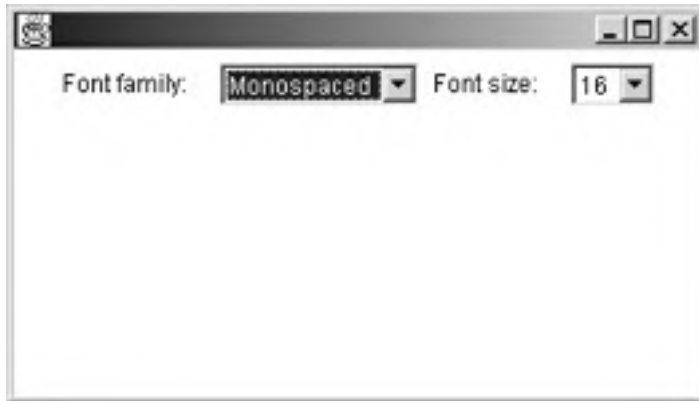


Figure 15.12: Choices with labels

The figure shows a typical use of labels. They rarely appear in isolation. Most often, they are used to give information or instructions about adjacent components.

Menus

Menus have a lot in common with choices:

- They drop down to reveal options.
- The options are arranged vertically.
- They roll back up after a selection is made.

There are also some important differences:

- Menus are attached to a frame's boundary, whereas choices occupy the interior.
- Menus are hierarchical. They can contain submenus, which can contain sub-submenus, and so on. Choices are linear.
- Commercial-grade sites are expected to have menus, which are expected to follow certain conventions. There are no such expectations or conventions for choices.

To insert menus into a frame, follow these steps:

1. Create a menu bar.
2. Create the menus.
3. Attach the menus to the menu bar.
4. Attach the menu bar to the frame.

These steps are all straightforward. The following code creates a frame with a single menu, labeled File. The menu contains three options: Open..., Close, and Exit.

```
1. import java.awt.*;
2.
3. class FrameWithSimpleMenu extends Frame
4. {
5.     String[] options = { "Open...", "Close", "Exit" };
6.
7.     public FrameWithSimpleMenu()
8.     {
9.         // Create the menu bar.
10.        MenuBar mbar = new MenuBar();
11.
12.        // Create the file menu.
13.        Menu fileMenu = new Menu("File");
14.        for (int i=0; i<options.length; i++)
15.            fileMenu.add(options[i]);
```

```
16.
17.     // Populate the menu bar.
18.     mbar.add(fileMenu);
19.
20.     // Attach the menu bar to the frame.
21.     setMenuBar(mbar);
22.
23.     setSize(200, 200);
24. }
25.
26. public static void main(String[] args)
27. {
28.     (new FrameWithSimpleMenu()).setVisible(true);
29. }
30. }
```

Line 10 creates a menu bar. In Java, a frame may have at most one menu bar, to which all menus must be attached. Line 13 creates a File menu. The string passed to the `Menu` constructor appears in the menu bar. Line 15 adds options to the menu. Line 18 attaches the menu to the menu bar. Menus appear on the bar in the order of attachment, from left to right. Finally, line 21 attaches the menu bar to the frame. The result is shown in [Figure 15.13](#).



Figure 15.13: A menu in a menu bar

The `Menu` class has a method called `addSeparator()`, which inserts a horizontal separator bar. You can rewrite the loop at lines 14-15 in the previous example, adding a separator bar between the `Close` and `Exit` items:

```
for (int i=0; i<options.length; i++)
{
    fileMenu.add(options[i]);
    if (i == 1)
        fileMenu.addSeparator();
}
```

The result is shown in [Figure 15.14](#).



Figure 15.14: A menu with a separator

In this example, you've created a menu and added items to it using the `add()` method, passing strings as method arguments. To create a hierarchical menu, add a menu instead of a string:

```
1. import java.awt.*;
2.
3. class FrameWithSubmenu extends Frame
4. {
5.     public FrameWithSubmenu()
6.     {
7.         MenuBar mbar = new MenuBar();
8.
9.         Menu subSubMenu = new Menu("Subsub");
10.        subSubMenu.add("This");
11.        subSubMenu.add("That");
12.        Menu subMenu = new Menu("Sub");
13.        subMenu.add("Here");
14.        subMenu.add("There");
15.        subMenu.add(subSubMenu);
16.        Menu fileMenu = new Menu("File");
17.        fileMenu.add("Open...");
18.        fileMenu.add("Close");
19.        fileMenu.add(subMenu);
20.
21.        mbar.add(fileMenu);
22.        setMenuBar(mbar);
23.
24.        setSize(200, 200);
25.    }
26.
27.    public static void main(String[] args)
28.    {
29.        (new FrameWithSubmenu()).setVisible(true);
30.    }
31. }
```

The GUI appears in [Figure 15.15](#).



Figure 15.15: Hierarchical menus

You can nest menus within menus within menus as much as you want, but don't get carried away. The more complicated your menu structure is, the more difficult it will be for users to find important menu items.

There are many industry-standard conventions that govern the use of menus in GUIs. These are the result of extensive psychological research, as well as many years of practical usage. Here are a few guidelines that are easy to follow:

- There should always be a File menu, and it should occupy the leftmost position in the menu bar. The items New, Open..., and Close should, if present, appear in that order. New should be the first item in the File menu. The Exit item should always be present and should be the last item in the File menu.
- If the application has an Edit menu, it should immediately follow the File menu. The Edit menu should support functions such as Cut, Copy, and Paste.
- If a Help menu is present, it should occupy the rightmost position menu bar.
- Any menu item that causes a new frame or dialog box to be displayed should have three dots following its label. This explains why Open menu items, which typically display file selection dialogs, appear as Open... The three-dots notation is called an *ellipsis*.

These guidelines should be followed in appropriate situations. It isn't necessary to follow them when you're writing code to solve exercises, but keep them in mind whenever you are writing code that is at least moderately complicated and will be used by other people. This book's animated illustrations are all moderately complicated. They are much bigger than exercises and much smaller than commercial applications. They all follow these guidelines.

Text Fields

A text field is a component that displays a single line of text. Unlike labels, text fields respond to keyboard input. To create a text field, use one of the following constructors:

- `TextField(String contents)`
- `TextField(int numColumns)`
- `TextField(String contents, int numColumns)`

The first version creates a text field that is just wide enough to accommodate its contents, which are specified by the string argument. The second version creates a blank text field that is wide enough to accommodate a string of `numColumns` characters. The width is only approximate, since most characters have varying widths when rendered in most fonts. The third version is like the second version, but the text field's contents are initialized to `contents`.

The following application creates two text fields for entering a first and last name. The Last Name text field uses a large non-default font:

```
import java.awt.*;

class TFs extends Frame
{
    public TFs()
    {
        setLayout(new FlowLayout());

        add(new Label("First Name: "));
        TextField tf = new TextField("Livia", 10);
        add(tf);
        add(new Label("Last Name: "));
        tf = new TextField("Soprano", 12);
        Font font = new Font("Monospaced", Font.PLAIN, 24);
        tf.setFont(font);
        add(tf);

        setSize(550, 200);
    }

    public static void main(String[] args)
    {
        (new TFs()).setVisible(true);
    }
}
```

Figure 15.16 shows the GUI.



Figure 15.16: Two text fields

As you can see from the figure, there is something dissonant about having two related text fields with two unrelated fonts. The GUI would be much improved if both fields used the same font, but it illustrates an important point: Text fields can grow to accommodate their fonts. For the moment, we will simply attribute this behavior to the mysterious layout manager, with a promise of a full explanation in the second half of this chapter.

Text Areas

A text area is like a text field, but it can display multiple lines of text. If its contents exceed its height, it can automatically display scrollbars.

The most useful `TextArea` constructor is

```
TextArea(int numRows, int numColumns)
```

The constructor creates a text area with `numRows` rows and `numColumns` columns. Caution: The order of the constructor's arguments might seem backwards. Generally, we are used to specifying first a width and then a height (for example, in the various drawing methods of the `Graphics` class). But `numRows` is a specification of height, and `numColumns` is a specification of width. If you get confused about which comes first, you might try to create a tall, narrow text area and end up with a short, broad one. We say that the dimensions of the text area are specified in *row major* order, which just means that the number of rows comes first.

Here is a very simple program that creates a text area:

```
import java.awt.*;

class TAInnaFrame extends Frame
{
    public TAInnaFrame()
    {
        setLayout(new FlowLayout());
        TextArea ta = new TextArea(10, 30);
        add(ta);
        setSize(550, 220);
    }
}
```

```
}  
  
public static void main(String[] args)  
{  
    (new TAINnaFrame()).setVisible(true);  
}  
}
```

The code is simple, but the text area is not. All we did was create a 10-by-30 text area, as shown in [Figure 15.17](#).



Figure 15.17: A text area

The text area's contents can be changed, either under program control or by user input. A little bit of typing results in [Figure 15.18](#).



Figure 15.18: Multiple checkbox groups

As more text is entered, the contents become taller than the component. When this happens, the text area automatically installs scroll bars, as shown in [Figure 15.19](#).



Figure 15.19: A text area with scroll bars

To add text to a text area programmatically, use the `append()` method, which takes a string argument. The string is added to the end of the component's contents. Caution: Text areas do not automatically word-wrap. If you want to add text on a new line, your new text should start with a newline character (`\n`).

Scrollbars

A checkbox has two states, true and false. A choice has several states, one state for each item that might be selected. Both component types are good for situations where users have a limited range of choices. Scrollbars are input devices that come into play when the range of choices is broad. They are most commonly seen in word processors and Web browsers, where they are used to specify the vertical position of a document.

The `java.awt.Scrollbar` class has two main constructors:

- `Scrollbar()`
- `Scrollbar(int orientation)`

The first version creates a vertical scrollbar. The second version creates a scrollbar that is either horizontal or vertical, depending on the value of the `orientation` argument. The class defines two static ints, called `Scrollbar.HORIZONTAL` and `Scrollbar.VERTICAL`. So to construct a horizontal scrollbar, you would call `new Scrollbar(Scrollbar.HORIZONTAL)`.

The following code creates two scrollbars, one in each orientation:

```
import java.awt.*;

class TwoBars extends Frame
{
    public TwoBars()
    {
        setLayout(new FlowLayout());
        add(new Scrollbar()); // Vertical
        add(new Scrollbar(Scrollbar.HORIZONTAL));
        setSize(200, 200);
    }

    public static void main(String[] args)
    {
        (new TwoBars()).setVisible(true);
    }
}
```

There is nothing very exciting about this code. Unfortunately, there is also nothing very exciting about the GUI it creates, as you can see from [Figure 15.20](#).



Figure 15.20: A pair of disappointing scrollbars

Neither of the scrollbars is long enough to be useful. A decent vertical scrollbar should be taller, and a decent horizontal scrollbar should be wider. You can't be effective at setting a component's height or width unless you know about layout managers. Fortunately, we have completed our survey of components, so let's move on.

Layout Managers

The `java.awt` package contains a very useful class called `Container`. Generally, you don't instantiate this class directly. Rather, you instantiate its various subclasses, which include `Frame`. All containers have the following properties:

- They are rectangular.
- They can contain other components, including other containers.
- They use layout managers to determine the locations and positions of the components they contain.

The great thing about layout managers is that you don't have to think about the details of component layout. Each layout manager class imposes a different layout policy on the container it manages. All you have to do is become familiar with the various layout policies available to you. The layout manager will take care of the details.

To change a container's layout policy, you construct an instance of the desired layout manager class. Then you call the container's `setLayout()` method, passing in the layout manager. All the code examples in this chapter have used frames, and the layout manager for a frame is something called a *border layout manager*. The border layout policy is completely inappropriate to what we wanted to do, but a different manager, the *flow layout manager*, is perfect.

This explains why every example constructed an instance of `FlowLayout` and then passed the instance into a `setLayout()` call. Usually this was done in a single line:

```
setLayout(new FlowLayout());
```

To understand layout policies, and therefore to understand why we used flow layout so extensively, you have to understand the concept of *preferred size*. Every component has a preferred size, which a layout manager can either honor or ignore. For components that have text, such as buttons and checkboxes, the preferred size is just large enough to accommodate the component's text. For components without text, the preferred size is arbitrary, which usually is not very good. The preferred size of a scrollbar, for example, is 15x50 pixels.

The Flow Layout Manager

The flow layout manager always honors the preferred size of its container's components. Every component in every figure in this chapter has been its preferred size, because every frame has used a flow layout manager.

When a container uses a flow layout manager, its contained components appear from left to right in the order they were added to the container. There is a gap of five pixels between adjacent components. The cluster of components appears at the top of the container and is centered horizontally. (Horizontal centering is the default. There are other options. See Exercise 6.)

The following code creates three components and uses a flow layout manager to position them in a frame:

```
import java.awt.*;

class SimpleFlow extends Frame
{
    public SimpleFlow()
    {
        setLayout(new FlowLayout());
        add(new Label("ABCDEFGH"));
        add(new Button("Hello"));
        Font f = new Font("SansSerif", Font.BOLD, 24);
        Button btn = new Button("Goodbye");
        btn.setFont(f);
        add(btn);
        setSize(300, 200);
    }

    public static void main(String[] args)
    {
        (new SimpleFlow()).setVisible(true);
    }
}
```

[Figure 15.21](#) shows the code's GUI. Notice how the components are spaced evenly and centered horizontally.



Figure 15.21: Flow layout manager

[Figure 15.22](#) shows the same GUI, after the frame has been made wider. The cluster is still centered.



Figure 15.22: Wider

And [Figure 15.23](#) shows the GUI one last time. Now the frame is too narrow to fit all three components. When this happens, the flow layout manager makes another row. If the frame were even narrower, there would be yet another row.



Figure 15.23: Narrower

You can configure flow layout managers to place their clusters at the left or right of their containers, rather than in the center. You do this by passing an `int` into the `FlowLayout` constructor. If the `int` is `FlowLayout.LEFT`, the cluster will appear at the left; if the `int` is `FlowLayout.RIGHT`, the cluster will appear at the right. [Figure 15.24](#) shows the three-component GUI of the current example, with the `setLayout()` line changed to

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```



Figure 15.24: Left-aligned

The Flow Lab animated illustration lets you experiment with components that are managed by a flow layout manager. To start the application, type `java flow.FlowLab`. [Figure 15.25](#) shows the initial screen.





Figure 15.25: Flow Lab

Flow Lab lets you select a left-, center-, or right-aligning layout manager. You can add buttons, checkboxes, text fields, and labels. Experiment with Flow Lab until you have a good feel for how the layout manager arranges its components. Select left alignment. Add components of a uniform size until the top row is full and a second row is created. How many components are in the top row? Does the number of components in the top row change when you select center or right alignment?

Notice that your selection of layout alignment affects only the main region of the display; the two control panels at the top of the display are not affected. It seems the layout manager you selected is only responsible for part, not all, of the frame. You will see how this is done a little later, in the "Panels" section. But first you have another layout manager to learn about.

The Border Layout Manager

The border layout policy is bizarre at first glance. It only makes sense after you learn what it's good for. So if your reaction to the next few paragraphs is, "This is weird," congratulations. You're right on track.

A border layout manager partially honors and partially ignores the preferred size of its container's components. A preferred size consists of two dimensions: width and height. A border layout manager might ignore one of a component's preferred dimensions while honoring the other. Or both preferred dimensions might be ignored. They are never both honored.

A container that uses a border layout manager may not contain more than five components. The layout manager divides the container into five regions, named North, South, East, West, and Center. Each region may be occupied by zero or one components.

The component in the North region is placed at the top of its container. The component's height is its preferred height; its width is the entire width of the container. [Figure 15.26](#) shows a horizontal scrollbar in the North region of a frame.



Figure 15.26: Scrollbar at North

Here is the code that produced [Figure 15.26](#):

```
1. import java.awt.*;
2.
3. class BarAtNorth extends Frame
4. {
5.     public BarAtNorth()
6.     {
7.         Scrollbar bar=new Scrollbar(Scrollbar.HORIZONTAL);
8.         add(bar, "North");
9.         setSize(200, 100);
10.    }
11.
12.    public static void main(String[] args)
13.    {
14.        (new BarAtNorth()).setVisible(true);
15.    }
16. }
```

The constructor does not call `setLayout()`, because you want to use a border layout manager, which is the default for a frame. In other words, the right kind of layout manager is already there.

Look at line 8. When you add components to a container that uses a border layout manager, you have to pass a second argument to the `add()` method. This is a string that must be North, South, East, West, or Center.

The component at South is attached to the bottom of the container. Otherwise, it is treated like the component at North. Its preferred height is honored, and its width is the entire width of the container.

[Figure 15.27](#) shows a frame with a horizontal scrollbar at North and a text field at South.



Figure 15.27: North and South occupied

The code that produced [Figure 15.27](#) is almost identical to the code that produced [Figure 15.26](#). The difference is that this code

has the following lines before the `setSize()` call:

```
TextField tf = new TextField("Hello");  
add(tf, "South");
```

As you might guess, the components at East and West are attached to the right and left edges of their container, respectively. Their preferred widths are honored. Their heights are the height of the container... almost. They extend all the way up to the top of the container, unless there is a component at North. In that case, they only extend to the bottom of the North component. Similarly, if there is no component at South, the East and West components can extend all the way down to the bottom of the container. But if there is a component at South, the East and West components extend down just to the top of the South component.

There are many combinations of the presence or absence of North or South or East or West components, but [Figure 15.28](#) should make things clear. In the figure, there are components at North, East, and West.



Figure 15.28: North, East, and West occupied

There is no component at South, so the two buttons extend down all the way to the bottom of the container. Since the scrollbar occupies North, the buttons do not extend all the way to the top of the container. They defer to North, extending up to the bottom of the scrollbar. Notice how the buttons have different preferred widths as a result of their different fonts.

So much for North, South, East, and West. The component at Center, if there is one, occupies all the territory that is left over after all other components have been sized and positioned. The white region in [Figure 15.28](#) is the area where there are no components, so the white background of the frame is visible. If the frame had a component at Center, that component would fill the white region exactly. In [Figure 15.29](#), a text area has been added at Center.



Figure 15.29: North, East, West, and Center occupied

Here is the code that produced [Figure 15.29](#):

```
1. import java.awt.*;
2.
3. class NEAndW extends Frame
4. {
5.
6.     public NEAndW()
7.     {
8.         Scrollbar bar=new Scrollbar(Scrollbar.HORIZONTAL);
9.         add(bar, "North");
10.        Button btn = new Button("Me West");
11.        add(btn, "West");
12.        btn = new Button("Me East");
13.        btn.setFont(new Font("Serif", Font.PLAIN, 50));
14.        add(btn, "East");
15.        TextArea ta = new TextArea();
16.        add(ta, "Center");
17.        setSize(600, 400);
18.    }
19.
20.    public static void main(String[] args)
21.    {
22.        (new NEAndW()).setVisible(true);
23.    }
24. }
```

Notice the `TextArea` constructor on line 15. This version is different from the one you were introduced to, where you passed in arguments to specify the number of rows and columns. The no-args version used here is for situations where the text area's size will be determined by the layout manager, so there is no need for you to specify a size.

Panels

Panels are components that divide containers into regions that are smaller and more manageable. The `java.awt.Panel` class extends `java.awt.Container`, so every panel has its own layout manager. You can think of panels as rectangular components that can contain other components, including panels. These in turn can include panels, and so on, so it is possible to create a complex layered hierarchical GUI.

[Figure 15.30](#) shows a frame whose South component is a panel containing three buttons. The panel's only other component is a text area at Center.

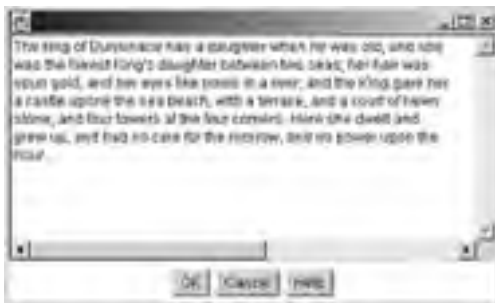


Figure 15.30: A panel in a frame

Here is the code that created [Figure 15.30](#):

```
1. import java.awt.*;
2.
3. class PanelInFrame extends Frame
4. {
5.     public PanelInFrame()
6.     {
7.         Panel pan = new Panel();
8.         pan.add(new Button("OK"));
9.         pan.add(new Button("Cancel"));
10.        pan.add(new Button("Help"));
11.        add(pan, "South");
12.        TextArea ta = new TextArea();
13.        add(ta, "Center");
14.        setSize(400, 250);
15.    }
16.
17.    public static void main(String[] args)
18.    {
19.        (new PanelInFrame()).setVisible(true);
20.    }
21. }
```

When you use panels, you make a lot of calls to the `add()` method of various containers. It's important to keep track of what is being added to what. On lines 8-10, the buttons are added to the panel. On line 11, the panel is added to the frame. On line 13, the text area is added to the frame.

The code has no `setLayout()` calls. The default layout manager for panels is flow. This is confusing, because the default manager for frames is border layout. You have to remember which container type defaults to which layout policy. But in practice, the defaults are usually what you want, so you don't often have to call `setLayout()`.

The flow layout manager makes more sense when you know about panels. Most GUI-based applications consist of a frame that contains a control area and a work area. For example, all Web browsers have a control panel at the top of the display with buttons for going forward, back, home, and so on. Below the control panel is the Web page viewing area. Generally, at the bottom is a status message. When you enlarge the browser, you don't want more space for the controls or the status message; you want a bigger Web page viewing area. The same holds true for most word processors, painting programs, and indeed most programs in general. When the user resizes, it is the main work area below the control area that should do most of the growing.

This is exactly the behavior that you get when you use a frame with a panel at North and some kind of work area at Center. The panel is attached to the top of the frame, and is as wide as the frame. It is as tall as it needs to be to accommodate the components it contains. (That's how the preferred height of a panel is defined.) When the frame becomes wider or narrower, the panel's components are repositioned automatically. When the frame becomes higher or shorter, it is the work area and not the panel that grows or shrinks. At the end of the [next chapter](#), after you have learned how to detect input activity from components, you will work through a final project whose GUI consists of a panel at North and a work area at South.

The Layout Lab animated illustration lets you experiment with hierarchical combinations of containers, layout managers, and components. Layout Lab is designed to let you play with layout ideas without going through the effort of writing code to implement your ideas. To start the program, type `java layout.LayoutLab`. You will see the display shown in [Figure 15.31](#).



Figure 15.31: Layout lab

Initially, the display displays a representation of a frame named Frame0. If you want to change the frame's properties, including its layout manager, click on the Frame0 button. You will see the dialog box shown in [Figure 15.32](#).



Figure 15.32: Layout lab's frame editing dialog

Make sure the frame's layout manager is set to Border. Then dismiss the edit dialog by clicking its Apply button. Now add a component to the frame. Click on the + button. You will see a small dialog that lets you choose a button, a scrollbar, a checkbox, a text field, or a panel. Select Panel, and then click the Apply button. The main window will now look like [Figure 15.33](#).

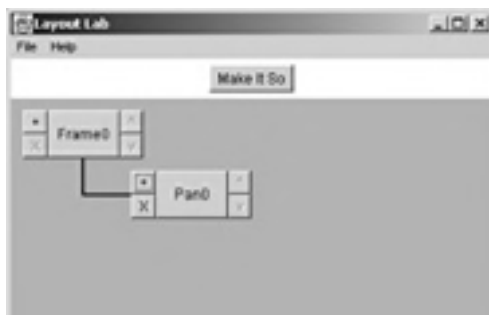




Figure 15.33: Layout Lab with an added panel

Now click on the Pan0 button to edit the properties of the new panel. Since the panel is inside the frame, which uses a border layout manager, one of the panel's properties is its region within the frame (North, South, East, West, or Center). Select South and then click the Apply button.

Now it's time to put a few buttons in the panel. In the main screen, click on the + button. When the little component-chooser dialog appears, select Button and then click Apply. Now the main window will look like [Figure 15.34](#).

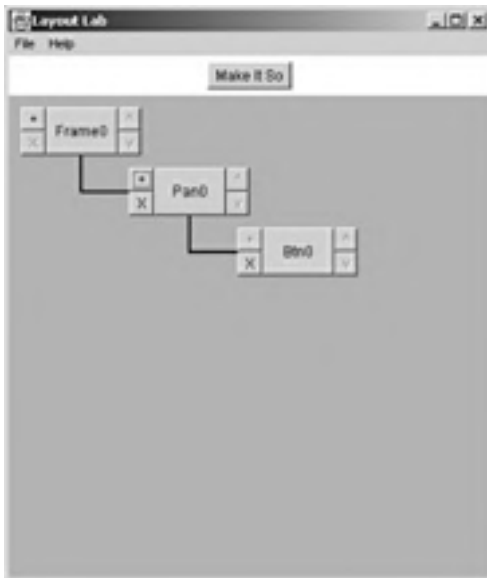


Figure 15.34: A button in a panel in a frame

Edit the button by clicking on Btn0 in the main window. You will see a dialog box that lets you edit the button's location, position, font, and text. Set both X and Y to 500. Set the font to something conspicuously non-default, like SansSerif 36-point bold italic. Set the label to whatever text you like, and click the Apply button.

Now you have created a description of a slightly complicated hierarchical GUI: a button in a panel at the South of a frame. To see what the GUI really looks like, click the Make It So button. You will see a frame that looks like [Figure 15.35](#).





Figure 15.35: Layout Lab makes it so

The button is definitely not 500x500 pixels in size. Is the button's property dialog broken? No. Remember that the panel is using a flow layout manager, which honors the button's preferred size. As you can see in [Figure 15.34](#), the button's preferred size is much smaller than 500x500. In fact, it is just large enough to accommodate the button's text.

Now experiment with layout lab. Try adding more buttons, or other kinds of components, to the panel. Add a panel to the frame's Center. Choose a layout manager for the new panel, add components, including a panel, and add components to *that*. If you want to get rid of a component, click on its X button in the main display. (If the component is a panel, all its contents will be deleted as well. You aren't allowed to delete the frame.) You can click the ^ and v buttons to change the ordering of components in their container.

Play with Layout Lab until you feel comfortable with the idea of components inside a panel that is inside a panel that is inside a frame.

Other Layout Managers

Before we leave the topic of layout managers, it is appropriate to mention that there are other options besides flow and border. The `java.awt` package provides three other managers, called `CardLayout`, `GridLayout`, and `GridBagLayout`. All three are beyond the scope of an introductory book, but you should know that they exist so that you can investigate them if you ever decide you need them.

`CardLayout` allows only one component to be seen at any time. `GridLayout` organizes its container into a grid of rows and columns; each component occupies a single grid location. `GridBagLayout` also creates rows and columns, but it provides many more options than `GridLayout` does.

Several other layout managers (`BoxLayout`, `OverlayLayout`, and `SpringLayout`) are part of the `javax.swing` package. Swing is an alternative to the AWT toolkit. Its components are much more sophisticated than those of AWT.

You can create your own layout manager class. To do this, you implement the `java.awt.LayoutManager` interface. It isn't especially hard once you get the hang of it. The interface only has five methods, and several of them are trivial. Many of this book's animated illustrations display Java source code that is mostly text, with a few scattered text fields or choices that allow you to configure the source code. The data chain lab in [Chapter 13](#) did this. This kind of layout cannot be achieved with any of the standard layout managers, so a new layout manager class was created.

There is one last layout manager option, and it is offered with caution. You can call `setLayout(null)` to operate with no layout manager at all. Then it is your responsibility to set the size and location of every component. You do this by calling the following methods on the components:

`void setLocation(int x, int y)` Sets the component's location (upper-left corner) to (x, y).

`void setSize(int width, int height)` Sets the component's size to width-by-height.

`void setBounds(int x, int y, int width, int height)` Sets the component's location to (x, y) and its size to width-by-height.

The following code uses no layout manager. It creates a 300-by-300 button and positions it at (40, 40):

```
import java.awt.*;

class NullLayout extends Frame
{
    public NullLayout()
    {
        setLayout(null);
        Button btn = new Button("Cancel");
        btn.setSize(300, 300);
        btn.setLocation(40, 40);
        add(btn);
        setSize(400, 400);
    }

    public static void main(String[] args)
    {
        (new NullLayout()).setVisible(true);
    }
}
```

Figure 15.36 shows the GUI.



Figure 15.36: No layout manager

The Layout Lab animated illustration lets you set any container's layout manager to None. Do this to the frame, and add two buttons labeled OK and Cancel. Edit each button's position and size until you like what you see. Get a feel for the ease or difficulty of this task.

The no-layout-manager strategy should be used with caution. As Figure 15.36 shows, it is easy to create a GUI with components of inappropriate size or location. Moreover, when your container has more than a very few components, it is unlikely to look good when the user resizes it.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Suppose you use the following code to create a checkbox:

```
Checkbox cbox = new Checkbox("Ok", true);
```

What is the checkbox's state after you click on it 20,000 times?

2. In the "Checkboxes" section of this chapter, the `Boats` application is 30 lines long. The code isolates literal strings in an array near the top of the listing. You saw how this approach, along with the use of a loop to create the checkboxes, results in more maintainable code. Rewrite the code to eliminate the loop and the string array. In place of the loop in the constructor, just create three checkboxes one by one. How many lines of code does your new application have?
3. This is an extension of Exercise 2. Suppose you need to change the `Boats` application so that instead of offering three sizes (small, medium, and large), it offers ten (rubber duck, sponge, tiny, small, kinda small, medium, kinda large, large, huge, titanic). How does this affect the size of the code as it appears in the "Checkboxes" section of this chapter? How does it affect the size of the code that you wrote for Exercise 2?
4. Write an application that displays a frame with a menu bar. The bar should have the following menus:
 - An Edit menu with items Copy and Cut.
 - A File menu with items Close, Exit, and Open.
 - A Help menu with item Help. Assume that clicking on this item will display a helpful dialog.
 - A Whatever menu with items Stuff and Nonsense. The Nonsense item should be a submenu with items Ordinary Nonsense and Extreme Nonsense.Make sure that your GUI follows the guidelines listed at the end of the "Menus" section.
5. Write a program that creates a GUI that looks like the following illustration. The text in the text area should be set programmatically by a single call to the text area's `append()` method. The call should come directly after the text area is constructed.



6. Using the API page for `java.awt.FlowLayout`, determine how to create a flow layout manager that right-justifies its cluster of components rather than centering it.
7. The `java.awt.Component` class, which is a superclass of `java.awt.Button`, has a method called `setSize(int width, int height)`. The method's documentation says that it resizes the component so that its size is `width` times `height`.

What do you expect the following code to do? First, read the listing and decide on your answer. Then, type in the code and run it. Did you see what you expected to see?

```
import java.awt.*;

class Q7 extends Frame
{
    public Q7()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Abcde");
        btn.setSize(500, 500);
        add(btn);
        setSize(700, 700);
    }

    public static void main(String[] args)
    {
        (new Q7()).setVisible(true);
    }
}
```

8. This entire chapter has been about components that are installed inside containers. The [previous chapter](#) was about painting. What happens if a frame that contains components also has a `paint()` method that paints a part of the screen that is occupied by a component? Write a program that will reveal the answer.

Chapter 16: Events

Overview

Now you know how to create components and lay them out in a GUI. The next step is to learn about *events*, which are the mechanism by which components inform us that they have been used.

A component that does not send events is like a doorbell that does not ring. It looks okay, but it can't make anything happen. It has *look* but no *feel*. In this chapter, you will learn how to make components responsive. This will prepare you for [Chapter 17, "Final Project,"](#) where you will observe in detail a Java application that uses painting, components, events, and all the other programming techniques presented in this book.

Java's original event mechanism was quite limited. It was designed back when it was believed that Java would mostly be used to create applets on Web pages, where space would be limited and GUIs would be simple. It soon became evident, however, that Java was an excellent programming language for domains that had nothing to do with Web pages. As non-Web-based Java applications propagated, GUIs became more complicated, and the current event mechanism was introduced in release 1.1.

The new mechanism is *scalable*, which means it is useful and efficient over a broad range of complexity, from very elementary GUIs to extremely intricate ones. This comes at a price. The event mechanism is not simple. It isn't horribly complicated, but it does consist of several interacting pieces, and it might not make sense until you have seen all the pieces. But hang in there. It will all make sense soon, and when it does, you will have a powerful tool for creating full-fledged GUIs that have both look and feel.

Event-Driven Programs

GUI-based programs are fundamentally different from the applications you saw and wrote prior to [Chapter 14 \("Painting"\)](#). The earlier programs began execution at the beginning of the `main()` method, ran through the end of `main()`, and that was that. When `main()` was finished, the program was finished. The Java Virtual Machine ceased to exist, and you saw a new prompt in your console window.

Now consider the behavior of an application with a GUI. The following code creates a frame that contains a button and displays a blue circle:

```
import java.awt.*;

public class Xxxx extends Frame
{
    Xxxx ()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Push Me");
        add(btn);
        setSize(300, 300);
    }

    public void paint(Graphics g)
    {
        g.setColor(Color.blue);
        g.fillOval(100, 100, 100, 100);
    }

    public static void main(String[] args)
    {
        (new Xxxx()).setVisible(true);
    }
}
```

[Figure 16.1](#) shows the GUI.

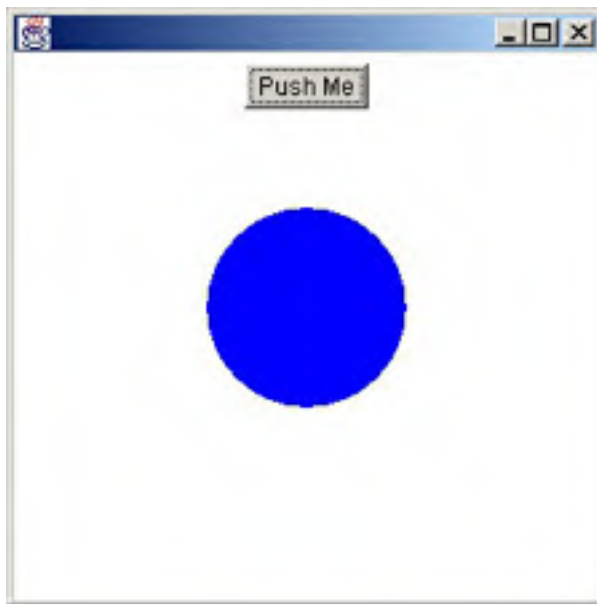


Figure 16.1: A GUI waiting for events

There is something mysterious about this code's behavior, and indeed about the behavior of all the GUI-based applications you saw in [Chapters 14](#) and [15](#). In fact, there are two mysteries, and perhaps you have already wondered about them.

- The `main()` method consists of a single line. After the instance of class `Xxxx` has been constructed and made visible, one would expect the program to terminate. This is not the case. The JVM continues to run (but doing what?) until someone types Ctrl+C into the console window. It is only then that the frame vanishes and the console is ready to accept a new command.
- The application does not call `paint()` anywhere. The method is implemented but never invoked. But something somewhere must have called it, because the circle is right there in the middle of the frame. Who called `paint()`?

Something within the Java Virtual Machine seems to be keeping an eye on things on our behalf, calling `paint()` at the right time and keeping the GUI alive after `main()` finishes. You could almost say the JVM has multiple personalities: One personality to run `main()`, and one to take care of the mysterious GUI behavior.

In fact, the JVM has the computer equivalent of multiple-personality disorder. Don't worry! For computers it's a good thing, because the personalities function together harmoniously. We say that the JVM is *multithreaded*, which means it is capable of

performing more than one task at a time. Each task is called a *thread*. Java's threading capabilities are powerful and intricate. You can write applications that create several or many threads, each performing its own task. Creating your own threads and maintaining harmony among them is far beyond the scope of this book. In order to understand GUI event processing, you don't need to know any details about creating or managing threads. But you do need to know a little bit about what threads are. As you'll see in the [next section](#), my own exposure to threads began many years ago in a barbershop.

Threads

I was eight years old. I was waiting my turn to get my hair cut, and I was reading a Superman comic book. The Man of Steel was entertaining some kids at an orphanage by playing ping-pong against himself. He would serve the ball, and then run at super-speed to the other side of the table to return the ball. Then he would run back to the original side, and so on. Each time he changed sides, he ran so fast that nobody could see him, and he ended up exactly where he had been before. This created the illusion of two identical Supermen.

Computers do something similar. They create the illusion of doing several things simultaneously by rapidly switching from one task to another, many times each second, like Superman running from one side of the ping-pong table to the other. The individual tasks are called *threads*.

Thread support is built into the Java language and the JVM. When you run an application, even a very simple one that just prints out a message from `main()`, there are actually two threads at work. One of these is called the *main thread*. Its job is to execute your `main()` method. Until you began working with GUIs, all behavior of all the applications you wrote came from the main thread.

The second thread is called the *garbage collection thread*. Recall from [Chapter 6, "Arrays,"](#) that Java's garbage-collection feature recycles memory from objects and arrays when they can be used no longer. This recycling happens while your program is executing. In other words, the main thread and the garbage collection thread operate simultaneously. You don't have to do anything special to make the garbage collection thread work; it is created automatically by the JVM.

Another thread that is created automatically as part of the JVM infrastructure is the *event dispatch thread*, also known as the *GUI thread*. It is not present in all applications; it appears only in applications with GUIs. The Event dispatch thread knows when the display needs to be redrawn and calls `paint()` at the appropriate moment. As you will see later in this chapter, it is the Event dispatch thread that knows when components have been activated and calls the appropriate methods in the appropriate objects.

The presence of an Event dispatch thread affects the life cycle of the JVM. If an application has no GUI, the JVM terminates when the main thread finishes its work. However, if an Event dispatch thread is present, the JVM continues to run after the main thread is done. The JVM remains in existence until the Event dispatch thread terminates. Typically, this happens when the Event dispatch thread executes a `System.exit()` call.

It is easy to imagine what the JVM is doing while the main thread is alive. Mostly, the JVM is executing the application's bytecode, but now and then the garbage collection thread recycles some memory. But what about a GUI application, where `main()` calls the constructor of a `Frame` subclass, calls `setVisible()` on the constructed object, and then is done? At this point the frame is on the screen, just sitting there. You saw this in numerous examples in the [previous chapter](#). If the frame is doing nothing, and `main()` has terminated, what is the JVM doing?

The answer is: Absolutely nothing! The Event dispatch thread is lurking in the background, waiting for the user to do something that requires attention. For example, if the frame becomes covered by another frame and is subsequently uncovered, the Event dispatch thread will call `paint()` so that the screen can be updated. It is also the job of the Event dispatch thread to notice when user input has occurred, and to respond appropriately by making certain method calls to certain objects.

We say that Java GUI programs are *event-driven*. This means that after some initialization, the programs only act in response to user input. An *event* is a single unit of user input. In the [next section](#), you will learn about Java's simplest type of event.

Action Listeners and Action Events

Java uses many types of events. The simplest is the action event, which is used by buttons and several other components to indicate that simple user input activity has occurred. The other event types are slightly more complicated than action events, but they are used in analogous ways. The nice thing about Java's event mechanism is that once you've learned how to handle one kind of event, it's easy to handle the other kinds.

Every button has a list of objects that are interested in being notified when the button is pushed. These objects are the button's *action listeners*. In general, a *listener* is an object that should be notified when a component is stimulated in some way.

Not all objects are eligible to be a button's action listener. An action listener must implement the `java.awt.event.ActionListener` interface. Note that this interface lives in the `java.awt.event` package, along with all the other classes and interfaces that make up Java's event mechanism. So a GUI application is likely to use the following two import lines:

```
import java.awt.*;
import java.awt.event.*;
```

The first line imports all the component classes; the second imports the event-related classes and interfaces.

The `java.awt.event.ActionListener` interface defines a single method:

```
public void actionPerformed(ActionEvent e);
```

When a button is pressed, the Event dispatch thread constructs an instance of `java.awt.event.ActionEvent`. This is a very simple class that contains a small amount of information about the button activity. Then the Event dispatch thread calls the `actionPerformed()` method of each of the button's action listeners, passing the instance of `ActionEvent` as the method call's argument.

When a button is constructed, its list of action listeners is empty. This explains why none of the buttons created in the example code in the [previous chapter](#) actually caused anything to happen. To add an action listener to a button's list, call the button's `addActionListener()` method, passing as an argument the listener to be added. The listener must implement the `ActionListener` interface.

Here is a class that implements the interface, and so is eligible to be a button's action listener:

```
import java.awt.event.*;

class SimpleActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("The button was pushed.");
    }
}
```

The following code creates a button that uses an instance of `SimpleActionListener` as its action listener:

```
import java.awt.*;

public class UsesListener extends Frame
{
    UsesListener ()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Push Me");
        SimpleActionListener sal = new SimpleActionListener();
        btn.addActionListener(sal);
        add(btn);
        setSize(300, 100);
    }

    public static void main(String[] args)
    {
        (new UsesListener()).setVisible(true);
    }
}
```

[Figure 16.2](#) shows the GUI.



Figure 16.2: A button that sends events

The important thing to notice about [Figure 16.2](#) is that there is nothing important to notice. The button looks perfectly ordinary. There is nothing to tell us that it has feel as well as look. But when you push it, the following message appears in your console:

The button was pushed.

Congratulations! You have now seen your first example of a GUI that has both look and feel.

In addition to the `addActionListener()` method, the `Button` class also has a `removeActionListener()` method, which can be called when a listener no longer wants to get called when the button is pushed.

Note In practice, buttons rarely have multiple action listeners, and `removeActionListener()` is rarely called. In most cases, a button has a single action listener that is added just after the button is constructed and is never removed.

The procedure for writing code with a button that responds to user input can be summarized as follows:

1. Construct a button.
2. Create a listener class that implements the `ActionListener` interface.
3. Construct an instance of your listener class.
4. Call the button's `addActionListener()` method, passing in the instance of your listener class.

The Simple Event Lab animated illustration lets you experiment with buttons and listeners without writing code. Start the program by typing `java events.SimpleEventLab`. You will see the display shown in [Figure 16.3](#).



Figure 16.3: Simple Event Lab: initial screen

The program lets you create simulated buttons and listener classes. You can create simulated instances of the simulated listener classes, click on the buttons, and observe how calls are made to the listeners.

Begin by creating some buttons. Click on Add Button three times. You will see things that look somewhat like buttons, as shown in [Figure 16.4](#).



Figure 16.4: Simple Event Lab with simulated buttons

Now create a (simulated) listener class. In real life, you would do this by writing a class that implements `ActionListener`. In Simple Event Lab, you do it by clicking on Create Listener Class... in the lower part of the frame. The button label ends with dot-dot (officially called *ellipsis*). As you learned in the [previous chapter](#), this means that the button causes a new frame or dialog

box to appear. Indeed, clicking the button brings up a dialog box that lets you choose the name of the class. After you dismiss the dialog, a picture of the class appears at the bottom of the screen, as shown in [Figure 16.5](#).



Figure 16.5: Simple Event Lab with a listener class

The figure shows that a listener class called `GoodListener` has been created. Create your own class, choosing any name you like.

Now it's time to create an instance of the listener class. Click on the picture of the class. You will see a pop-up menu that lets you instantiate the class or delete it. Choose Construct Instance. A simulated instance of the class will appear below the simulated buttons in the main screen, as you can see in [Figure 16.6](#).

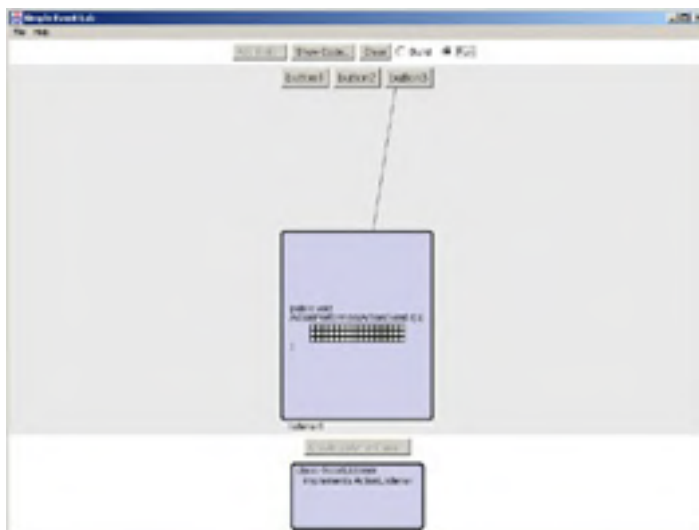


Figure 16.6: Simple Event Lab with a listener object

You can click on the picture of the listener object to change its name or to delete it.

Up to this point, you have simulated the first three steps listed earlier in this section. You have created buttons, you have created a listener class, and you have constructed an instance of the listener class. Now it's time to register the listener object as an action listener of one of the buttons.

Click on one of the simulated buttons. You will see a pop-up menu that invites you to add an action listener or delete the component. Choose Add Action Listener. The cursor will turn into crosshairs. As you move the mouse over the listener object, the object's outline will be highlighted, indicating that you are over a valid listener for the button. Click on the listener. You will see a line connecting the button to the listener.

Now the fun begins. Click the Run button at the top of the screen. The simulated buttons will turn into real buttons, as shown in [Figure 16.7](#).

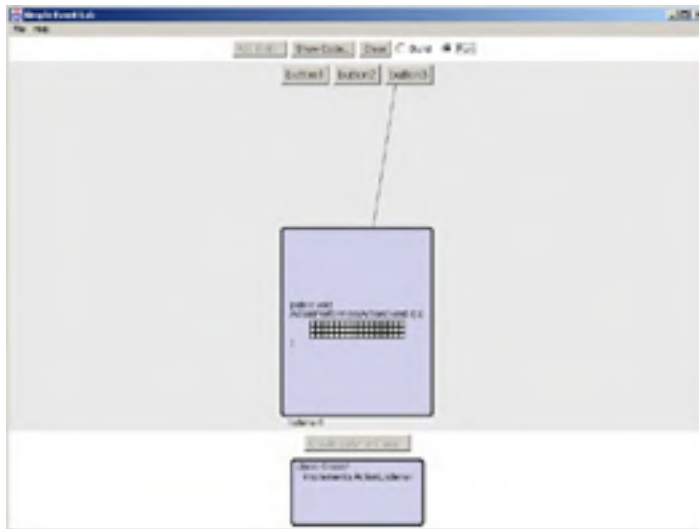


Figure 16.7: Simple Event Lab continued

Now click the button that you connected to the action listener object. The program will show a call being made to the listener's `actionPerformed()` method. The yellow ball represents the `ActionEvent` object.

Click on the Clear button to remove all simulated components, listener classes, and listener objects. Now that you have a clean slate, see if you can repeat the process of connecting a button to a listener without looking at this page.

Experiment with multiple listener classes and multiple listener objects. Can a single listener object be an action listener for more than one button? Can a button have more than one action listener? What does the Show Code... button do?

Getting Information from an Action Event

In the [previous section](#), you were asked to use Simple Event Lab to determine whether a single listener object can be an action listener for more than one button. The answer is yes, as shown in [Figure 16.8](#).

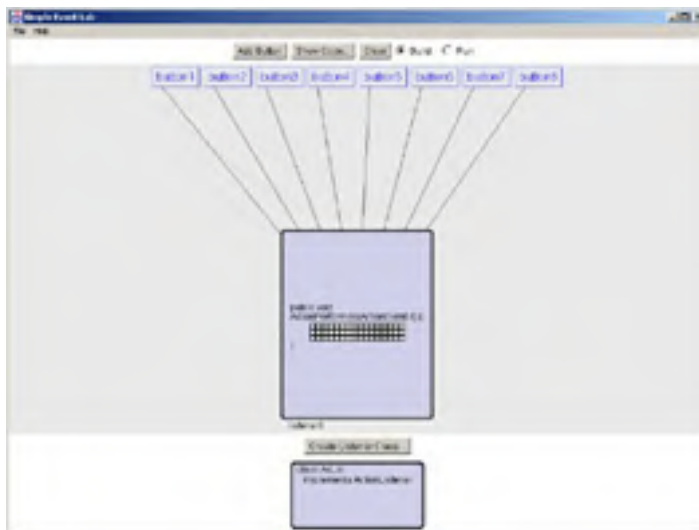


Figure 16.8: One listener object for many buttons

But now there is a problem. Obviously, the code needs to respond differently to different buttons. How does the listener's `actionPerformed()` method know which button was clicked?

The answer is found inside the method's argument. The `ActionEvent` class has a `getSource()` method that returns the button that was clicked. Many `actionPerformed()` methods have a structure that is similar to the following:

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == okButton)
        doOkStuff();
    else if (e.getSource() == cancelButton)
        doCancelStuff();
    else if (e.getSource() == applyButton)
        doApplyStuff();
}
```

The method determines which button was used and responds accordingly. For this to work, the method has to have access to references to the three buttons. The simplest way to make this happen is to put `actionPerformed()` in the frame subclass that creates the buttons. Make sure the frame subclass declares that it implements `ActionListener` (no problem, since it has an

`actionPerformed()` method). Finally, when the buttons are created, the frame subclass itself is registered as their action listener. It looks like this:

```
1. import java.awt.*;
2. import java.awt.event.*;
3.
4. public class ListeningFrame extends Frame
5.     implements ActionListener
6. {
7.     private Button  okButton, cancelButton, applyButton;
8.
9.     ListeningFrame()
10.    {
11.        setLayout(new FlowLayout());
12.        okButton = new Button("Ok");
13.        okButton.addActionListener(this);
14.        add(okButton);
15.        cancelButton = new Button("Cancel");
16.        cancelButton.addActionListener(this);
17.        add(cancelButton);
18.        applyButton = new Button("Apply");
19.        applyButton.addActionListener(this);
20.        add(applyButton);
21.        setSize(300, 100);
22.    }
23.
24.    public void actionPerformed(ActionEvent e)
25.    {
26.        if (e.getSource() == okButton)
27.            doOkStuff();
28.        else if (e.getSource() == cancelButton)
29.            doCancelStuff();
30.        else if (e.getSource() == applyButton)
31.            doApplyStuff();
32.    }
33.
34.    public static void main(String[] args)
35.    {
36.        (new ListeningFrame()).setVisible(true);
37.    }
38. }
```

The `implements ActionListener` statement makes the `ListeningFrame` class eligible to be an action listener for buttons. Lines 13, 16, and 19 register `this` as each button's listener. Recall that `this` is a reference to an object that owns the code being executed. In other words, it's the instance of `ListeningFrame` that is being constructed. The `doOkStuff()`, `doCancelStuff()`, and `doApplyStuff()` methods are omitted.

Here is another example that uses the same design structure. The program plays a version of the game Nim. This game is played by placing 10 coins in a pile. Each player in turn takes one, two, or three coins. The player who takes the last coin is the winner. The GUI consists of four buttons: Take 1, Take 2, Take 3, and Quit. As each player takes a coin, the code prints out the number of remaining coins. [Figure 16.9](#) shows the GUI.

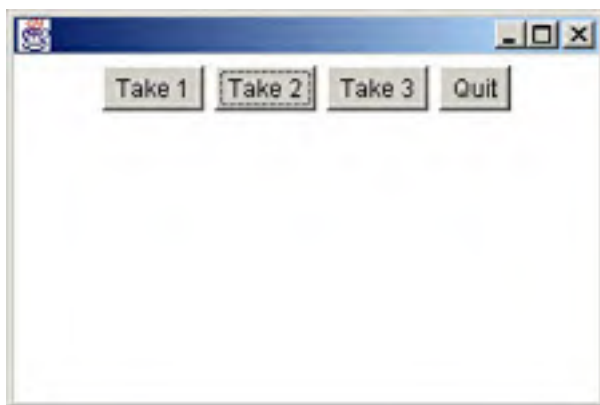


Figure 16.9: Simple Nim GUI

Here's the code:

```
import java.awt.*;
import java.awt.event.*;

public class SimpleNim extends Frame
    implements ActionListener
{
    private Button  btn1, btn2, btn3, quitBtn;
    private int     nCoins;

    SimpleNim()
    {
        nCoins = 10;
        setLayout(new FlowLayout());
    }
}
```

```
        btn1 = new Button("Take 1");
        btn1.addActionListener(this);
        add(btn1);
        btn2 = new Button("Take 2");
        btn2.addActionListener(this);
        add(btn2);
        btn3 = new Button("Take 3");
        btn3.addActionListener(this);
        add(btn3);
        quitBtn = new Button("Quit");
        quitBtn.addActionListener(this);
        add(quitBtn);
        setSize(300, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == quitBtn)
            System.exit(0);

        if (e.getSource() == btn1)
            nCoins -= 1;
        else if (e.getSource() == btn2)
            nCoins -= 2;
        else if (e.getSource() == btn3)
            nCoins -= 3;
        System.out.println(nCoins + " left.");
    }

    public static void main(String[] args)
    {
        (new SimpleNim()).setVisible(true);
    }
}
```

The `actionPerformed()` method first determines if the event source was the Quit button. If so, `System.exit()` is called to terminate the program. In event-driven programming, calling `System.exit()` in response to user input is the appropriate way to end a program. If the event came from one of the Take buttons, the coin count `nCoins` is decremented and the remaining value is printed out.

This application works as an example of how to process events, but it is certainly no improvement over a pile of coins. (Unless you don't have 10 coins. But if you don't have 10 coins, you probably can't afford a computer.) The situation points out an important principle of GUI design, which is violated all too often on the World Wide Web: Only create a GUI if it makes life better.

In the [next section](#), you will see the last example improved on in several ways. You might not think the final version is better than a pile of coins, but you will certainly find it an improvement over the original version. And, more importantly, you will learn some important techniques for creating useful GUIs.

Improving the GUI

In this section, the `SimpleNim` application will be improved in three stages. To keep life simple, the Nim Lab program on your CD-ROM gives you easy access to all four versions (the original and the three improvements). To run Nim Lab, type `java events.NimLab`. You will see the display shown in [Figure 16.10](#).

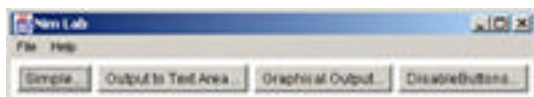
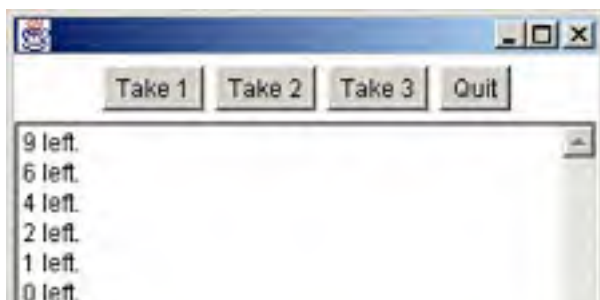


Figure 16.10: Nim Lab

Each improvement will illustrate a general principle of GUI design. The first principle is that the results of user input activity should appear near where the activity happened. In this way, cause and effect are visually related. (The cause is the input, and the effect is the resulting change to the screen.) In the `SimpleNim` version, you clicked buttons in the GUI, but your output appeared at the console from which you ran the program. This is inconvenient, because you have to keep moving your eyes back and forth.

It would be better if the output could happen in the GUI. For this, you will use a text area. The `TextArea` class has a method called `append()` that appends text the component's contents, so let's modify the `actionPerformed()` method so that it calls `append()` rather than `System.out.println()`.

[Figure 16.11](#) shows the GUI after a game has been played.



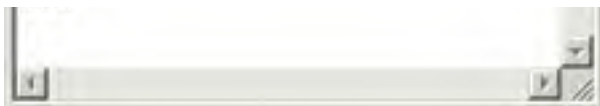


Figure 16.11: Nim, with output to a text area

Here is the code:

```
import java.awt.*;
import java.awt.event.*;

public class TextAreaNim extends Frame
    implements ActionListener
{
    private Button    btn1, btn2, btn3, quitBtn;
    private TextArea  ta;
    private int       nCoins;

    TextAreaNim()
    {
        nCoins = 10;

        Panel controls = new Panel();
        btn1 = new Button("Take 1");
        btn1.addActionListener(this);
        controls.add(btn1);
        btn2 = new Button("Take 2");
        btn2.addActionListener(this);
        controls.add(btn2);
        btn3 = new Button("Take 3");
        btn3.addActionListener(this);
        controls.add(btn3);
        quitBtn = new Button("Quit");
        quitBtn.addActionListener(this);
        controls.add(quitBtn);
        add(controls, "North");

        ta = new TextArea(40, 20);
        add(ta, "Center");
        setSize(300, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == quitBtn)
            System.exit(0);

        if (e.getSource() == btn1)
            nCoins -= 1;
        else if (e.getSource() == btn2)
            nCoins -= 2;
        else if (e.getSource() == btn3)
            nCoins -= 3;
        ta.append(nCoins + " left.\n");
    }

    public static void main(String[] args)
    {
        (new TextAreaNim()).setVisible(true);
    }
}
```

The frame uses a border layout manager. There is a panel (`controls`) at North containing the buttons. The text area is at Center. Thus, when you make the frame bigger (try it!), most of the new space goes to the text area.

The original line

```
System.out.println(nCoins + " left.");
```

has been replaced by

```
ta.append(nCoins + " left.\n");
```

Notice the newline character (`\n`) in the new version. When you call `System.out.println()`, a newline is printed automatically. This does not happen when you call `append()` on a text area, so you have to provide your own newline.

This version is definitely an improvement. You no longer have to look up to do input and look down to read output. But the output is pure text.

The next principle of GUI design that we will apply is this: Show me, don't tell me. Our next improvement will be to draw coins on the screen, rather than displaying text that merely tells you about coins. This is not a book on graphic design, so the coins will just be filled circles. But the code will show what you could do if you were working with a graphics designer who provided you with code for painting exquisitely detailed coins.

Figure 16.12 shows the initial state of the new version.

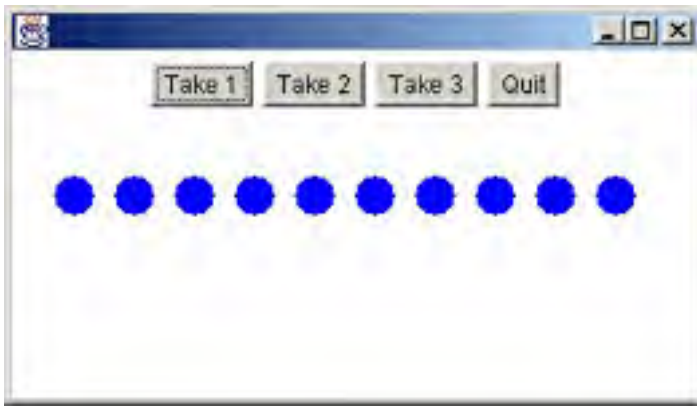


Figure 16.12: Nim with graphical output

Figure 16.13 shows the GUI after a few coins have been taken.

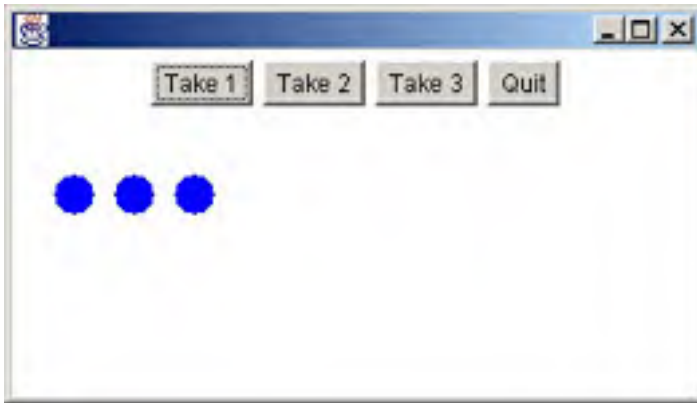


Figure 16.13: Nim with graphical output, game in progress

Figures 16.12 and 16.13 dramatically show that pictures are better than words. Here's the code:

```
import java.awt.*;
import java.awt.event.*;

public class GraphicOutputNim extends Frame
    implements ActionListener
{
    private Button    btn1, btn2, btn3, quitBtn;
    private int      nCoins;

    GraphicOutputNim()
    {
        nCoins = 10;

        Panel controls = new Panel();
        btn1 = new Button("Take 1");
        btn1.addActionListener(this);
        controls.add(btn1);
        btn2 = new Button("Take 2");
        btn2.addActionListener(this);
        controls.add(btn2);
        btn3 = new Button("Take 3");
        btn3.addActionListener(this);
        controls.add(btn3);
        quitBtn = new Button("Quit");
        quitBtn.addActionListener(this);
        controls.add(quitBtn);
        add(controls, "North");
        setSize(350, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == quitBtn)
            System.exit(0);

        if (e.getSource() == btn1)
            nCoins --;
        else if (e.getSource() == btn2)
            nCoins -= 2;
        else if (e.getSource() == btn3)
            nCoins -= 3;
        repaint();
    }
}
```

```
    }  
  
    public void paint(Graphics g)  
    {  
        int x = 25;  
        int y = 85;  
  
        g.setColor(Color.blue);  
        for (int i=0; i<nCoins; i++)  
        {  
            g.fillOval(x, y, 20, 20);  
            x += 30;  
        }  
    }  
  
    public static void main(String[] args)  
    {  
        (new GraphicOutputNim ()).setVisible(true);  
    }  
}
```

The text area is gone. The line at the end of `actionPerformed()`, which originally called `System.out.println()` and then called the text area's `append()` method, is now the following:

```
repaint();
```

The `repaint()` method causes two things to happen:

1. The frame's interior is cleared to its background color. (Components contained in the frame are not affected.)
2. A call is made to the frame's `paint()` method.

Whenever you want a display to be refreshed in response to GUI input, calling `repaint()` is the best approach. It would be beyond the scope of this book to explain why. Here is an oversimplified explanation: When you call `repaint()`, eventually your `paint()` method will be called, at an appropriate time, with an appropriate `Graphics` argument. You never need to call `paint()` directly. You are always better off calling `repaint()` and letting the environment call `paint()`.

The `paint()` method uses a loop to draw the appropriate number of blue-filled circles, based on the value of `nCoins`. The variable `x` determines the horizontal position of each circle's bounding box. It is increased by 30 each time a circle is drawn. To see the program in action, run Nim Lab and select Graphical Output...

Now let's make one last improvement to enforce what is perhaps the most important GUI principle of all. If you pay attention to the other principles, you might create a great GUI. But if you ignore the most important principle, you will certainly create a poor GUI.

Here's the most important principle: A GUI should *never* let a user perform illegal input.

The latest Nim version violates this rule. To see this, run Nim Lab and select Graphical Output.... Click the Take 3 button three times. Now there is only one coin left, but the GUI will let you take two or three coins. This should not be allowed. You also should not be allowed to take three coins if there are two coins left.

There are two ways to make illegal input impossible. At the appropriate time, the buttons can be either removed or disabled. Removing the buttons may sound like a good idea (after all, you can't push a button that isn't there), but extensive research has shown that users are uncomfortable with GUIs whose components pop in and out of existence. This approach creates too much movement in the peripheral field of vision. The commonly accepted technique is to disable components that should not be used. The components are still visible, but they are unresponsive. A disabled component has a slightly different appearance. It is somewhat grayer than its enabled counterpart. [Figure 16.14](#) shows two buttons. The first is enabled, the second is disabled.



Figure 16.14: Enabled and disabled buttons

In the previous version of the Nim GUI, the Take buttons are enabled only if there are enough coins left. [Figure 16.15](#) shows the program when one coin remains.



Figure 16.15: Nim with disabled buttons

Notice that the Take 2 and Take 3 buttons are disabled. To enable or disable any component, call its `setEnabled()` method. The method takes a boolean argument: `true` to enable, `false` to disable. Here's the code:

```
import java.awt.*;
import java.awt.event.*;

public class DisablingNim extends Frame
    implements ActionListener
{
    private Button    btn1, btn2, btn3, quitBtn;
    private int      nCoins;

    DisablingNim()
    {
        nCoins = 10;

        Panel controls = new Panel();
        btn1 = new Button("Take 1");
        btn1.addActionListener(this);
        controls.add(btn1);
        btn2 = new Button("Take 2");
        btn2.addActionListener(this);
        controls.add(btn2);
        btn3 = new Button("Take 3");
        btn3.addActionListener(this);
        controls.add(btn3);
        quitBtn = new Button("Quit");
        quitBtn.addActionListener(this);
        controls.add(quitBtn);
        add(controls, "North");
        setSize(350, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == quitBtn)
            System.exit(0);

        if (e.getSource() == btn1)
            nCoins -= 1;
        else if (e.getSource() == btn2)
            nCoins -= 2;
        else if (e.getSource() == btn3)
            nCoins -= 3;

        if (nCoins < 3)
            btn3.setEnabled(false);
        if (nCoins < 2)
            btn2.setEnabled(false);
        if (nCoins < 1)
            btn1.setEnabled(false);

        repaint();
    }

    public void paint(Graphics g)
    {
        int x = 25;
        int y = 85;

        g.setColor(Color.blue);
        for (int i=0; i<nCoins; i++)
        {
            g.fillOval(x, y, 20, 20);
            x += 30;
        }
    }

    public static void main(String[] args)
    {
        (new DisablingNim()).setVisible(true);
    }
}
```

The new code appears at the end of `actionPerformed()`:

```
if (nCoins < 3)
    btn3.setEnabled(false);
if (nCoins < 2)
    btn2.setEnabled(false);
if (nCoins < 1)
    btn1.setEnabled(false);
```

Further improvements to the GUI are possible. (See [Exercise 5](#) at the end of this chapter.)

But enough about Nim. At this point, you know how to respond to GUI input from buttons. It will be easy to move on to responding to other component types.

Team LIB

← PREVIOUS

NEXT →

Events from other Components

In [Chapter 15](#), you learned how to create a variety of component types:

- Buttons
- Check boxes
- Choices
- Labels
- Menus and menu items
- Scrollbars
- Text areas
- Text fields

Now you will learn how to respond to user input activity on each type of component. You already know how to respond to buttons. Labels do not send events. In the rest of this chapter, you will learn how to detect events from the other component types.

Check Boxes, Choices, and Item Events

In this section, you'll learn how to respond to activity from check boxes and choices. As a reminder, [Figure 16.16](#) shows a check box and a choice.

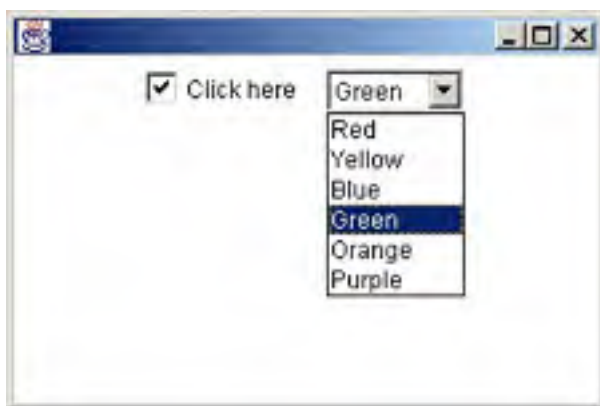


Figure 16.16: Check box and choice

Check boxes and choices don't have action listeners, but they have something similar: item listeners. An object that wants to be notified when activity happens in a check box or a choice must implement the `java.awt.event.ItemListener` interface. This interface defines one method:

```
public void itemStateChanged(ItemEvent e);
```

When a check box or choice is activated, an `itemStateChanged()` call is made to each of its item listeners. You can call `addItemListener(ItemListener x)` to add an item listener to a check box's or choice's list. You can call `removeItemListener(ItemListener x)` to remove an item listener from a check box's or choice's list.

Within an `itemStateChanged()` method, you can determine which component was activated by calling the `ItemEvent`'s `getSource()` method, just as you would call `getSource()` on an `ActionEvent` in an `actionPerformed()` method. (In fact, `ActionEvent`, `ItemEvent`, and the other event classes you will learn about in this chapter all inherit `getSource()` from a superclass that they all extend.)

The following code builds a GUI that contains a check box, a choice, and a text field. When the check box or the choice are activated, the text field displays an appropriate message.

```
import java.awt.*;
import java.awt.event.*;

public class CboxAndChoice extends Frame
    implements ItemListener
{
    private Checkbox    cbox;
    private Choice      ch;
    private TextField   tf;

    CboxAndChoice()
    {
        setLayout(new FlowLayout());
        cbox = new Checkbox("Click here");
        cbox.addItemListener(this);
        add(cbox);
```

```
        ch = new Choice();
        ch.add("Red");
        ch.add("Yellow");
        ch.add("Blue");
        ch.add("Plaid");
        ch.add("Paisley");
        ch.addItemListener(this);
        add(ch);

        tf = new TextField(25);
        add(tf);

        setSize(475, 75);
    }

    public void itemStateChanged(ItemEvent e)
    {
        if (e.getSource() == cbox)
            tf.setText("Checkbox: " + cbox.getState());
        else
            tf.setText("Choice: " + ch.getSelectedIndex());
    }

    public static void main(String[] args)
    {
        (new CboxAndChoice()).setVisible(true);
    }
}
```

The `itemStateChanged()` method calls the event's `getSource()` method to determine which component was activated. The `getState()` method of `Checkbox` returns `true` if the component is checked, and `false` if it is not checked. The `getSelectedIndex()` method of `Choice` returns the position (counting from 0) of the component's selected item.

Figure 16.17 shows the GUI.



Figure 16.17: Receiving events from a check box and a choice

By now, you probably get the feel of it. Components have lists of listeners. When the components are activated, method calls are made to the listeners.

That's about it. You'll probably have an easy time with the next several sections.

Text Fields and Text Areas

Text fields and text areas both send text events to text listeners. The events are sent each time a user types a keystroke. The `TextListener` interface defines one method:

```
public void textValueChanged(TextEvent e);
```

To add an object to a text field's or text area's list of text listeners, call the component's `addTextListener()` method, passing in the listener object.

Text fields (but not text areas) can also send action events to action listeners. This happens when the user presses the Enter key.

We won't work through a detailed code example, because if you understand how to handle action and item events, handling text events should be obvious. Instead, let's step back for a moment and look at the big picture.

A Java GUI consists of a number of components of various types. Each component may have zero, one, or multiple listeners for each event type that the component supports. When a component is activated, the Event dispatch thread calls the appropriate method of each listener.

The Event Lab animated illustration lets you experiment with multiple component, listener, and event types, without writing any code. Event Lab is an extension of Simple Event Lab. In addition to buttons, you can create check boxes, choices, and text fields. When you create a listener class, you select which listener interfaces it will implement. (Your choices are `ActionListener`, `ItemListener`, and `TextListener`. Remember that classes are allowed to implement more than one interface, so listener classes are allowed to implement more than one listener interface.)

Start the program by typing `java events.EventLab`. You control the program just as you did Simple Event Lab. Figure 16.18 shows Event Lab with a fairly complicated configuration.

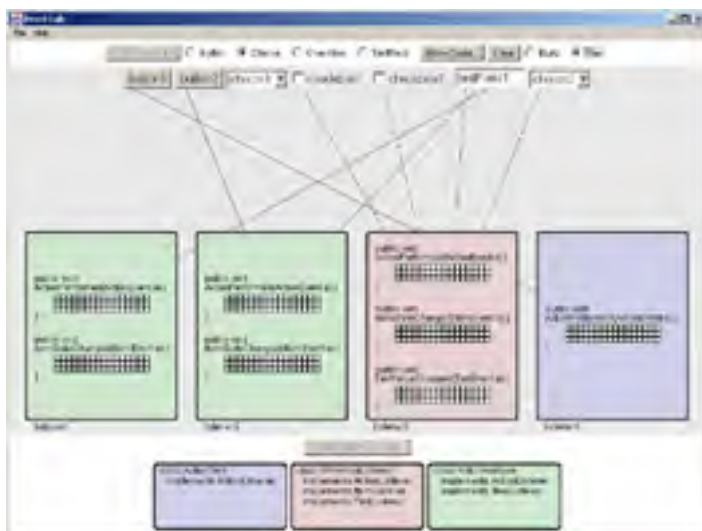


Figure 16.18: Event Lab

Configure Event Lab with your own complicated setup. Then click the Run button. The simulated components will become real. Activate your buttons, check boxes, choices, and text fields until you have a good feel for how various component types send various event types to various listeners in an event-driven GUI program.

Events from Menus

In [Chapter 15](#), you saw the simplest way to populate menus. It looked something like this:

```
Menu fileMenu = new Menu("File");
fileMenu.add("Open...");
fileMenu.add("Close");
...
```

The `add()` calls created individual menu items. That approach was good for showing you what Java menus look like, but there is a better way if you want to receive event notification from the menu items. The preceding code can be rewritten as follows:

```
Menu fileMenu = new Menu("File");
MenuItem openMI = new MenuItem("Open...");
fileMenu.add(openMI);
MenuItem closeMI = new MenuItem("Close");
fileMenu.add(closeMI);
...
```

Like buttons, menu items send action events to action listeners. So to add `menuListener` to `openMI`'s list of action listeners, you would call

```
openMI.addActionListener(menuListener);
```

There's no need to present a detailed example, because the code is so similar to the code you've already seen that handles action events from buttons.

Scrollbars and Adjustment Events

Scrollbars send adjustment events to adjustment listeners. Adjustment listeners implement the `java.awt.event.AdjustmentListener` interface. Once again, we have a listener interface that defines a single method:

```
public void adjustmentValueChanged(AdjustmentEvent e);
```

An object gets added to a scrollbar's listener list via a call to the `addAdjustmentListener()` method. The following code receives adjustment notification from a scrollbar, and reports the scrollbar's value to a text field. The code uses the `getValue()` method of the `Scrollbar` class. The return type is int:

```
import java.awt.*;
import java.awt.event.*;

public class BarAndTF extends Frame
    implements AdjustmentListener
{
    private Scrollbar bar;
    private TextField tf;

    BarAndTF()
    {
        bar = new Scrollbar(Scrollbar.HORIZONTAL);
        bar.addAdjustmentListener(this);
        add(bar, "North");
        Panel pan = new Panel();
        tf = new TextField(" ");
        pan.add(tf);
        add(pan, "South");

        setSize(300, 100);
    }
}
```

```
public void adjustmentValueChanged(AdjustmentEvent e)
{
    tf.setText("Value = " + bar.getValue());
}

public static void main(String[] args)
{
    (new BarAndTF()).setVisible(true);
}
}
```

The GUI is shown in [Figure 16.19](#).



Figure 16.19: Scrollbar and text field

Now you know how to respond to events from all the component types you learned about in [Chapter 15](#). In the [next chapter](#), which finishes this book, you will work through a detailed final project that draws from everything you have learned so far, from [Chapter 1](#) through the period at the end of this sentence.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Write a program that displays a frame. The frame's `paint()` method should draw something simple. The application should also maintain a count of the number of times `paint()` is called. This count should be printed out every time `paint()` is called. Execute your application, and use it to help determine whether `paint()` is called when:

The application starts up.

The frame is minimized/iconified.

The frame is restored to normal size after being minimized/iconified.

The frame is restored to normal size after being minimized/iconified.

The frame is moved.

The frame is partially covered by another frame.

The frame is uncovered.

2. Every Java thread is represented by an instance of the `java.lang.Thread` class. You can get a reference to the currently running thread by calling the `currentThread()` static method of the `Thread` class. Threads have names. The class has a method called `getName()`, which returns the name as a string. So you can print out the name of the current thread by calling

```
System.out.println(Thread.currentThread().getName());
```

Write a simple frame application that makes this call in its `main()` method and in its `paint()` method. Verify that `main()` and `paint()` are executed in different threads.

3. Write an application that adds the same action listener to a button *twice*. For example, if `myButton` is the button and `myListener` is the action listener, your code would contain the following lines:

```
myButton.addActionListener(myListener);  
myButton.addActionListener(myListener);
```

Your listener's `actionPerformed()` method should print out a message to tell you that it got called. If you press the button once, do you expect the message to be printed out once or twice? Run your application to see if you guessed right.

Of course, in real life there would never be a good reason for doing this. But you might do it by accident. For example, you might paste the line into your source code twice by accident. So it's good to know in advance what the symptom will be, so that you can recognize it and fix the problem if it ever comes up.

4. Suppose a class has an `actionPerformed()` method, as specified by the `ActionListener` interface, but the class does not state that it implements the interface. Can an instance of the class be used as a button's action listener?
5. Run Nim Lab by typing `java events.NimLab`. Select Disable Buttons... and play the game. This version is the result of three rounds of improvements made to the original program. What additional improvements can you suggest? Think about how the game could be modified to make the GUI easier and more natural.
6. The various event classes (`ActionEvent`, `ItemEvent`, etc.) all inherit the `getSource()` method from a superclass. Use the API pages to determine the name of that superclass.
7. Write an application with a GUI that contains a choice and a text area. When the choice is activated, a message should be written to the text area, stating the choice's selected index.

Suggested design: Your frame should contain a panel (at North) that contains the choice. The text area should be at South. If you need a guideline, the `TextAreaNim` program in the "Improving the GUI" section has a similar structure.

8. Write an application with a GUI that contains a text field and a text area. When the user presses the Enter key in the text field, the text field's contents should be copied into text area, followed by a newline character.

Your event-handling code will need to retrieve the contents of the text field. You do that by calling the text field's `getText()` method, which returns a string.

Suggested design: Your frame should contain a panel at North that contains the text field. The text area should go at Center.

Chapter 17: Final Project

You made it! With the presentation on event handling in the [previous chapter](#), you have finished your from-the-ground-up introduction to the Java programming language. You now know a *lot* about Java, and in this chapter you'll prove it. You will observe the development of a substantial programming project, and it will all make sense. The project will draw on the information you learned in every other chapter of this book. It includes a GUI that paints, uses components, and sends out events. Classes will be extended and interfaces will be implemented. Exceptions will be thrown and caught.

This chapter doesn't just walk you through a finished, polished program. That would be like dissecting an animal in high-school biology class. Seeing something grow and develop is better than studying something that's dead. So for each piece of the project, you will see not just the final product, but also the living process that culminates in the finished, polished program.

Description of the Project

We will create a GUI that displays Java source code in an easy-to-read format. The user will be able to choose any .java file. Most of the code will appear in black letters, but line comments and Java keywords will appear in different colors, to be specified by the user.

[Figure 17.1](#) shows the application in action. It is displaying one of the source files of the project.



Figure 17.1: Final Project

The Show lines check box draws horizontal lines to make the text more readable. [Figure 17.2](#) is the same code, with lines.

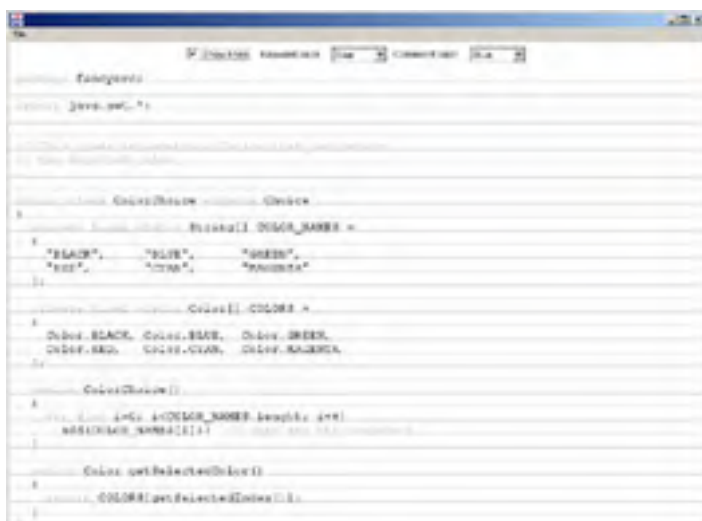


Figure 17.2: Final Project, with lines

The black-and-white figures don't communicate the dramatic effect of multicolored source code.

Now is a good time to run the actual project code, so you can get familiar with what you'll be doing in the rest of this chapter. Type `java fancysrc.FancySrcFrame`. You'll see the GUI controls shown in the figures, above a blank display area. To display

some code, select Open... in the File menu. You will see a file chooser. Use the chooser to select one of the source files on the CD-ROM, or one of your own Java source files.

Now let's get to work.

Team LIB

PREVIOUS **NEXT**

Building the Pieces

The overall structure of the project code will be familiar to you. We will create a subclass of `java.awt.Frame`, called `FancySrcFrame`, in package `fancysrc`. The frame will have a panel at North, containing various components. The `FancySrcFrame` class will be the event listener for all events from all components.

We will divide the work into five pieces. We will develop each piece in turn before assembling everything into the final product. The five pieces are:

- The menu
- The file-specification code
- The color-specification code
- The display area
- The painting code

Each piece of the project is discussed in its own section.

The File Menu

Over the years, the software industry has created a substantial number of conventions for interacting with GUI-based programs. Every GUI is different, but they can all be approached in the same way, with the same reasonable expectations. This is enormously beneficial to the community of software users (that's us), because it reduces the amount of time we have to spend learning to use a new program.

The automobile industry is in a similar situation. If you know how to drive a car, you pretty much know how to drive *every* car. If this were not the case, car rental would be even more stressful than it already is.

One of the standard GUI practices is to install a menu bar in every program's main frame. The leftmost menu is a File menu, whose last item is Exit. In our case, the File menu will have an Open... item. Here we won't worry about how to actually open a file. That's covered in the [next section](#), "Specifying a File." For the moment, our concern is to construct a menu bar with a File menu.

Building and responding to a menu requires techniques that were presented in [Chapters 15, "Components,"](#) and 16, ["Events."](#) The constructor for our main application class (`FancySrcFrame`, in package `fancysrc`) will build the menu.

When you write code that builds menus, you might find it helpful to draw a diagram like the one in [Figure 17.3](#).



Figure 17.3: Menu schematic

A menu schematic might be trivial for the project at hand, but it makes life much easier if you are creating complicated menu bars, with many menus and submenus. After you write your menu code, you can test all the menus to make sure they match your schematic.

[Chapter 15](#) presented a list of steps for building a menu structure:

1. Create a menu bar.
2. Create the menus.
3. Attach the menus to the menu bar.
4. Attach the menu bar to the frame.

Here is some code that builds the menu structure:

```
1. MenuBar mbar = new MenuBar();
2. Menu fileMenu = new Menu("File");
3. openMI = new MenuItem("Open...");
4. openMI.addActionListener(this);
5. fileMenu.add(openMI);
6. exitMI = new MenuItem("Exit");
7. exitMI.addActionListener(this);
8. fileMenu.add(exitMI);
9. mbar.add(fileMenu);
10. setMenuBar(mbar);
```

Line 1 creates a menu bar. Lines 2-8 create a menu. Line 9 attaches the menu to the menu bar, and line 10 attaches the menu bar to the frame. Assume the current class implements the `java.awt.event.ActionListener` interface, so `this` is a legal argument to the `addActionListener()` calls in lines 4 and 7.

The variables `mbar` and `fileMenu` are declared within the constructor. (Remember, all the preceding code goes in the

FancySrcFrame constructor.) However, `openMI` and `exitMI` will be declared as variables of the `FancySrcFrame` class. You'll see why shortly. Meanwhile, can you guess? (Hint: It has something to do with event handling.)

Now that we have a small chunk of code, let's test it. We'll embed it in a test class called `MenuTest`. This class implements `ActionListener`, so that lines 4 and 7 will compile. If the code works, we can later copy it verbatim from the test program into our final project code.

Here is the source for `MenuTest`:

```
package fancysrc;

import java.awt.*;
import java.awt.event.*;

class MenuTest extends Frame implements ActionListener
{
    private MenuItem openMI, exitMI;

    MenuTest()
    {
        MenuBar mbar = new MenuBar();
        Menu fileMenu = new Menu("File");
        openMI = new MenuItem("Open...");
        openMI.addActionListener(this);
        fileMenu.add(openMI);
        exitMI = new MenuItem("Exit");
        exitMI.addActionListener(this);
        fileMenu.add(exitMI);
        mbar.add(fileMenu);
        setMenuBar(mbar);
        setSize(250, 100);
    }

    public void actionPerformed(ActionEvent e)
    {
    }

    public static void main(String[] args)
    {
        (new MenuTest()).setVisible(true);
    }
}
```

The `actionPerformed()` method doesn't do anything, because for now we just want to check the structure of the menu. We are testing look, not feel. If you want to run `MenuTest`, it's on your CD-ROM. Just type `java fancysrc.MenuTest`. It looks like [Figure 17.4](#).



Figure 17.4: Teting the menu's look

The figure matches the menu schematic from [Figure 17.3](#), so apparently the code is good.

Now let's add code to respond to menu activation, so we can test feel as well as look. We just need to put some `println()` calls in the `actionPerformed()` method. Later we'll replace the calls with code that actually opens a file or exits the program.

Here's the new version of `actionPerformed()`:

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == openMI)
        System.out.println("OPEN Menu item");
    else
        // Must be the "Exit" menu item.
        System.out.println("EXIT Menu item");
}
```

When the test program is run and the two menu items are activated one after another, the output is

```
OPEN Menu item
EXIT Menu item
```

The output shows that the menu item action events are being handled correctly.

Implementing the exiting code is trivial. We just insert a call to `System.exit()`:

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == openMI)
        System.out.println("OPEN Menu item");
    else
        // Must be the "Exit" menu item.
        System.exit(0);
}
```

The test code is on your CD-ROM. If you want to run it, type `java fancysrc .MenuEventTest`. At this point the menu looks right, and its events are being handled properly, so we can move on to the question of how to open a file.

Specifying a File

Most computer programs modify data stored in files. This accounts for the prominent position of the File menu. Selecting a file for a program to process is a very common activity. You would expect file selection to be standardized in some way, and this is indeed the case.

Java provides a class called `java.awt.FileDialog`, which supports all the functionality needed to help users specify a file. The class is easy to use. As the name implies, it creates a *dialog box*. A dialog box is a window that is subordinate to its program's main frame, used for brief user interaction. When you delete a file or exit a program, and a box pops up to ask you if you're sure, you are looking at a dialog box.

Many dialog boxes are *modal*. A modal dialog box consumes all mouse and keyboard input to the program. This implies that you can't continue using the program until you have dealt with the dialog box and dismissed it. Most "Are you sure?" dialog boxes are modal. Java's file dialog box is also modal.

The `FileDialog` class shares a lot of behavior with the `Frame` class. This is not surprising, since the classes have a common superclass called `Window`, as shown in [Figure 17.5](#).

A glance at the API shows that `FileDialog` class has three constructors:

```
FileDialog(Frame parent)
FileDialog(Frame parent, String title)
FileDialog(Frame parent, String title,
           int mode)
```

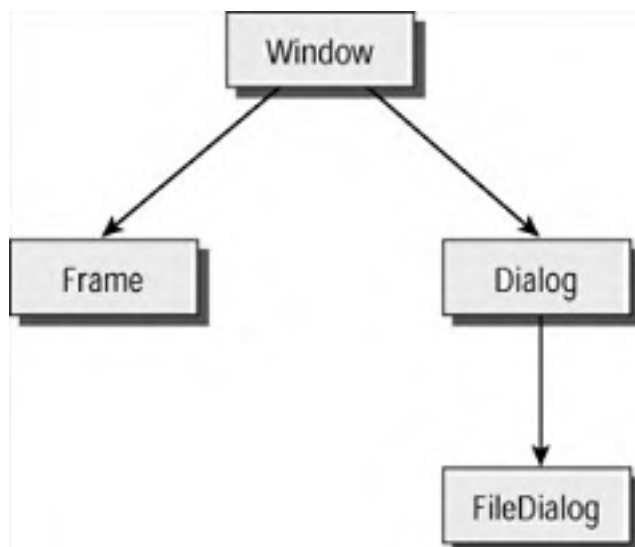


Figure 17.5: Window, Frame, and FileDialog

The `parent` argument is the frame over which the dialog box will appear. The `title` string determines what appears in the dialog box's title bar. The `mode` specifies whether the dialog box will be used for opening or saving a file. Opening is the default, so you don't have to worry about specifying the mode. (But see Exercise 1 at the end of this chapter.)

[Figure 17.6](#) shows a file dialog box, configured for opening:

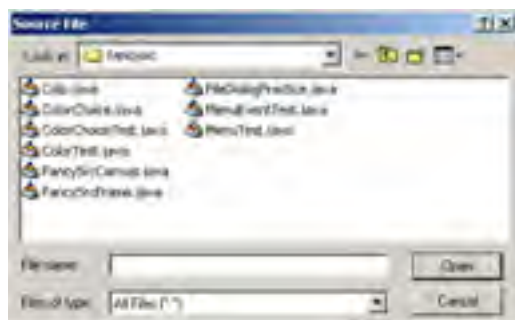


Figure 17.6: File dialog box configured for opening

Unlike frames, file dialog boxes are created with non-zero width and height, so you don't have to call `setSize()` on them. However, like frames, they are not visible until you call `setVisible(true)` on them. When you make this call, the dialog box appears, and the rest of the program's GUI refuses to accept mouse or keyboard input. Moreover, execution of your program pauses. Eventually, the user deals with and dismisses the dialog box. At this point, the rest of the GUI once more accepts input, and execution of your program continues from the line immediately following the `setVisible(true)` call.

The functionality is complicated, but using file dialog boxes is actually very simple. You construct your dialog box and, at the right moment, call `setVisible(true)` on it. The next line of code will not execute until a file has been specified (or the user has selected Cancel). There are two useful calls that you can then make on your dialog box, and both methods return strings:

getFile() The `getFile()` method returns the name of the file the user chose, or `null` if the dialog box was canceled.

getDirectory() The `getDirectory()` method returns the name of the chosen directory.

Whenever you learn about a new Java class, it's a good idea to write a practice program that creates an instance of the class and uses it in a way similar to the way you will later be using it in your program. That way you can experiment freely, and there is no danger that you will break your project accidentally by deleting or changing perfectly good code.

Here is a practice program that creates a file dialog box when a button is clicked:

```
package fancysrc;

import java.awt.*;
import java.awt.event.*;

class FileDialogPractice extends Frame implements ActionListener
{
    public FileDialogPractice()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Show me...");
        btn.addActionListener(this);
        add(btn);
        setSize(200, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        FileDialog dia = new FileDialog(this);
        dia.setVisible(true);
        String fileName = dia.getFile();
        if (fileName == null)
            System.out.println("You canceled the dialog.");
        else
            System.out.println("Your chose file " + fileName + "
                               in " + dia.getDirectory());
    }

    public static void main(String[] args)
    {
        (new FileDialogPractice()).setVisible(true);
    }
}
```

Notice the code in `actionPerformed()`:

1. `FileDialog dia = new FileDialog(this);`
2. `dia.setVisible(true);`
3. `String fileName = dia.getFile();`
- ...

After line 1 executes, processing does not move on to line 2 until the user has dismissed the dialog box. The `getFile()` call on line 3 returns `null` if the dialog box was canceled. If you want to try the test program, it's on your CD-ROM. To run it, type `java fancysrc.FileDialogPractice`.

What should the code do after the file has been specified? We don't know yet, but we will figure it out in good time. At this point, we have code to capture the user's desired input file. Before we worry about processing and displaying the file, let's turn our attention to the remaining GUI-related piece of the puzzle.

Specifying Colors

This section will look at the portion of the GUI that supports color selection. First you'll see a perfectly reasonable design: straightforward, but nothing fancy. Then the design will be improved in stages, ending with code that is elegant and reusable.

Let's start with what we know. We want to users to select from among a small number of colors. The colors must be dark enough that they can be read easily on a white background. That rules out yellow, pink, and several others. Let's settle on these:

- Black
- Blue
- Green
- Red

- Cyan
- Magenta

Cyan and magenta are marginal. For now, we'll include them. We can throw them out later on if they don't look good. If throwing them out proves to be difficult, that's an indication that our design wasn't very flexible.

Our users will have to select from these six colors... twice. Once for the keyword color, and once for the comment color. Of the components that you learned about in [Chapter 15](#), there are two that support making an exclusive selection from a small set of options: choices and radio buttons. We can rule out radio buttons because we would need 12 of them, compared to only two choices. If we used radio buttons, they would dominate the GUI, forcing the control area to be much larger than it needs to be, as shown in [Figure 17.7](#).



Figure 17.7: Too many radio buttons

For this situation, choices are much cleaner. Let's assume that our main application class will be called `FancySrcFrame` and will extend `Frame`. The class code will include the following declarations:

```
private Choice keywordChoice, commentChoice;
```

The choice components should be built in the `FancySrcFrame` constructor. One way to build them would be like this:

```
keywordChoice = new Choice();
keywordChoice.add("BLACK");
keywordChoice.add("BLUE");
keywordChoice.add("GREEN");
keywordChoice.add("RED");
keywordChoice.add("CYAN");
keywordChoice.add("MAGENTA");
commentChoice = new Choice();
commentChoice.add("BLUE");
commentChoice.add("BLACK");
commentChoice.add("GREEN");
commentChoice.add("RED");
commentChoice.add("CYAN");
commentChoice.add("MAGENTA");
```

This code can be improved, because every call appears twice. Whenever code is duplicated, consider the alternative of creating a method. The following code is much easier to read and more reliable:

```
keywordChoice = buildColorChoice();
commentChoice = buildColorChoice();
...
private Choice buildColorChoice()
{
    Choice c = new Choice();
    c.add("BLACK");
    c.add("BLUE");
    c.add("GREEN");
    c.add("RED");
    c.add("CYAN");
    c.add("MAGENTA");
    return c;
}
```

The new version is 13 lines long, compared to 14 in the original. That's not much of a difference, but later you might want to add a third color choice, and perhaps a fourth. In the old version, each additional color choice required seven lines, compared to only one line in the new version. Moreover, all choice components created by the `buildColorChoice()` method will be identical. With the original approach, each time you type the seven repeated lines, you introduce the possibility of a transcription error. Did you notice that in the first block of code, the second choice reverses the order of `BLACK` and `BLUE`?

An even cleaner approach uses an array of color names. The following would appear along with the other variables of the `FancySrcFrame` class:

```
private String[] colorNames =
{
    "BLACK", "BLUE", "GREEN", "RED", "CYAN", "MAGENTA"
};
```

Now the `buildColorChoice()` method is just the following:

```
private Choice buildColorChoice()
{
    Choice c = new Choice();
    for (int i=0; i<colorNames.length; i++)
        c.add(colorNames[i]);
    return c;
}
```

If you want to add or remove colors from the set of options, you just edit the contents of `colorNames`.

The choices will need an item listener. The logical candidate is the `FancySrcFrame` class. The `itemStateChanged()` method should cause the display to be repainted, using the new keyword or comment color. Somewhere (we don't need to decide where right now), some code will have to figure out which colors to use, based on the settings of the two choices. One way to do this would be to have a method that returns an instance of `Color`:

```
private Color getColorFromChoice(Choice c)
{
    int index = c.getSelectedIndex();
    if (index == 0)
        return Color.BLACK;
    else if (index == 1)
        return Color.BLUE;
    else if (index == 2)
        return Color.GREEN;
    else if (index == 3)
        return Color.RED;
    else if (index == 4)
        return Color.CYAN;
    else
        return Color.MAGENTA;
}
```

That certainly works, but there is a much cleaner way. First, we'll create an array of colors. For maximum readability, it should appear next to the `colorNames` array:

```
private Color[] colors =
{
    Color.BLACK, Color.BLUE, Color.GREEN,
    Color.RED, Color.CYAN, Color.MAGENTA
};
```

To determine the color indicated by a choice component, use the choice's selected index as an index into the `colors` array:

```
private Color getColorFromChoice(Choice c)
{
    int index = c.getSelectedIndex();
    return colors[index];
}
```

We have now worked out one piece of our design. We could go on to work out all our other design decisions, but before we blaze ahead, let's test what we have so far. If it doesn't work, we need to try again. If it works, we aren't committed to it. We reserve the right to improve on our color-specifying design later on.

Since color specification is the first code we will develop, our test will be simple. We don't yet know how we will select the file to be read, or paint lines on the screen, or paint source code on the screen in appropriate colors. So we'll create a program that just implements the color-specifying part of the GUI. To verify that the right colors are being returned from `getColorFromChoice()`, we'll just draw two squares in the frame. The square on the left will be the keyword color. The square on the right will be the comment color. [Figure 17.8](#) shows the GUI.

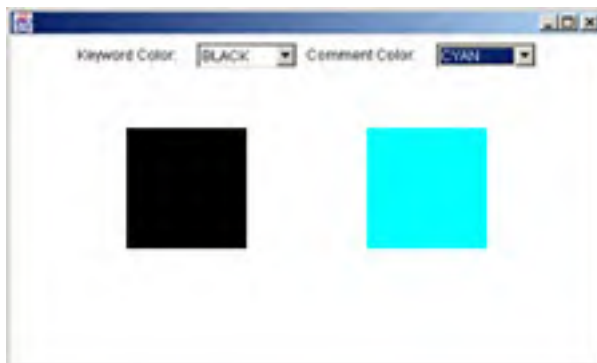


Figure 17.8: Testing color selection

Here's the code:

```
package fancysrc;

import java.awt.*;
import java.awt.event.*;

class ColorTest extends Frame implements ItemListener
{
    private String[] colorNames =
    {
        "BLACK",    "BLUE",    "GREEN",
        "RED",      "CYAN",    "MAGENTA"
    };

    private Color[] colors =
    {
        Color.BLACK, Color.BLUE, Color.GREEN,
        Color.RED,   Color.CYAN, Color.MAGENTA
    };
}
```

```
private Choice keywordChoice, commentChoice;

public ColorTest()
{
    setLayout(new FlowLayout());
    add(new Label("Keyword Color:"));
    keywordChoice = buildColorChoice();
    keywordChoice.addItemListener(this);
    add(keywordChoice);
    add(new Label("Comment Color:"));
    commentChoice = buildColorChoice();
    commentChoice.addItemListener(this);
    add(commentChoice);
    setSize(500, 300);
}

private Choice buildColorChoice()
{
    Choice c = new Choice();
    for (int i=0; i<colorNames.length; i++)
        c.add(colorNames[i]);
    return c;
}

private Color getColorFromChoice(Choice c)
{
    int index = c.getSelectedIndex();
    return colors[index];
}

public void itemStateChanged(ItemEvent e)
{
    repaint();
}

public void paint(Graphics g)
{
    Color keywordColor = getColorFromChoice(keywordChoice);
    g.setColor(keywordColor);
    g.fillRect(100, 100, 100, 100);
    Color commentColor = getColorFromChoice(commentChoice);
    g.setColor(commentColor);
    g.fillRect(300, 100, 100, 100);
}

public static void main(String[] args)
{
    new ColorTest().setVisible(true);
}
}
```

The class code begins with the arrays `colorNames` and `colors`. Notice how each name is aligned vertically with its corresponding color. It's a small touch that creates a visual relationship between the functionally related items.

The `itemStateChanged()` method just calls `repaint()`. Remember from [Chapter 16](#) that when you want to paint your display in reaction to user input, you shouldn't directly call `paint()`. Rather, you should call `repaint()`, which clears the display and then calls `paint()`. Our `paint()` method draws the two squares.

It works. If you want to try it, the code is on your CD-ROM. Just type `java fancysrc .ColorTest`.

We can't rest on our laurels yet. The code works, `ColorTest` proves it, but it isn't very object-oriented. The software that supports a single function (color selection) is spread throughout the class. The great thing about object-oriented programming is that it allows you to encapsulate related functionality. Let's see how to encapsulate color selection.

Think about the `Choice` class. Its `getSelectedIndex()` method returns an `int`. A lot of the code in `ColorTest` is devoted to converting that `int` to the corresponding color. Life would be a lot easier if `Choice` had a method called `getSelectedColor()`. Of course, no such method exists, because `Choice` is a general-purpose class intended for specifying colors, fonts, font sizes, names, countries, languages, or anything else that any programmer might think of. But we can subclass `Choice` to create a special-purpose class that does exactly what we want.

We will create a subclass called `ColorChoice`. The constructor will populate the component with the appropriate strings. The `colorNames` and `colors` arrays will go inside the new class, since no other code will need them. We will provide a `getSelectedColor()` method. The new class looks like this:

```
package fancysrc;

import java.awt.*;

public class ColorChoice extends Choice
{
    private static String[] colorNames =
    {
        "BLACK",      "BLUE",      "GREEN",
        "RED",        "CYAN",     "MAGENTA"
    };

    private static Color[] colors =
    {
```

```
        Color.BLACK, Color.BLUE, Color.GREEN,
        Color.RED,    Color.CYAN, Color.MAGENTA
    };

    public ColorChoice()
    {
        for (int i=0; i<colorNames.length; i++)
            add(colorNames[i]);
    }

    public Color getSelectedColor()
    {
        return colors[getSelectedIndex()];
    }
}
```

Notice that the two arrays have been declared as `static`. Remember that if a variable is static, there is only one copy of it, shared by all instances of the class. We know that there will be two instances of `ColorChoice`. There is no need to create two identical versions of the arrays, which is what would happen if they were not static. Each instance would have its own version. If the GUI changed later so that there were 25 color choices, there would be 25 identical versions of each array. Duplication of data is always something to be avoided. Here we avoid it by making the arrays static.

Testing the code is much easier. The complicated stuff is now in the `ColorChoice` class. The test code becomes the following:

```
package fancysrc;

import java.awt.*;
import java.awt.event.*;

class ColorChoiceTest extends Frame implements ItemListener
{
    private ColorChoice keywordChoice, commentChoice;

    public ColorChoiceTest()
    {
        setLayout(new FlowLayout());
        add(new Label("Keyword Color:"));
        keywordChoice = new ColorChoice();
        keywordChoice.addItemListener(this);
        add(keywordChoice);
        add(new Label("Comment Color:"));
        commentChoice = new ColorChoice();
        commentChoice.addItemListener(this);
        add(commentChoice);
        setSize(500, 300);
    }

    public void itemStateChanged(ItemEvent e)
    {
        repaint();
    }

    public void paint(Graphics g)
    {
        Color keywordColor = keywordChoice.getSelectedColor();
        g.setColor(keywordColor);
        g.fillRect(100, 100, 100, 100);
        Color commentColor = commentChoice.getSelectedColor();
        g.setColor(commentColor);
        g.fillRect(300, 100, 100, 100);
    }

    public static void main(String[] args)
    {
        new ColorChoiceTest().setVisible(true);
    }
}
```

The arrays are gone. The variables `keywordChoice` and `commentChoice` are now declared as type `ColorChoice`. We can call `addItemListener()` on them, just as if they were instances of `Choice`, because they inherit all the event-processing functionality of `Choice`. If you want to run the code, type `java fancysrc.ColorChoiceTest`. The GUI looks just like the earlier test GUI, so there's no need for a screenshot.

Now we can rest on our laurels! The source for class `ColorChoice` is less than 30 lines long, and look at what it can do:

- Look good.
- Behave exactly like a standard `Choice`.
- Be manipulated by a layout manager.
- Send out item events when activated.
- Report the selected color.

That's not a bad resume for such a small class. But we can't rest on our laurels all day. It's time to develop the rest of the code.

The Main Display Area

Most of the GUI work is now complete. We still have to create the check box that requests lines, but we'll get to that a bit later. Now we're going to step back and look at the big picture.

Our display will be involved a lot of painting. The painting examples you saw in [Chapter 14](#) all involved painting a frame. You saw subclasses of `java.awt.Frame` with specialized versions of the `paint()` method. Later you saw that when user input makes it necessary to revise the display, you should call your frame's `repaint()` method, which causes the screen to be cleared and the `paint()` method to be called.

As it happens, the `repaint()` mechanism works for certain other component types in addition to frames. There is a class called `java.awt.Canvas` that has no inherent appearance at all. If you construct a canvas and install it in a GUI, you won't see anything worth mentioning. That's okay, because you never actually put a canvas in a GUI. You create a subclass of `Canvas`, with a `paint()` method that draws whatever you want, and it is the subclass that you use in your GUI.

We will use a `Canvas` subclass, called `FancySrcCanvas`, for our main display area. The frame that contains everything will use its default `Border` layout manager. The control components (the Show lines check box and the two color choices) will go in a panel at North, and the canvas will be at Center, as shown in [Figure 17.9](#).



Figure 17.9: GUI layout

The `FancySrcCanvas` will need to redisplay itself whenever the user changes the file, the Show lines preference, or the keyword or comment color. The GUI code will detect all these changes. The `FancySrcCanvas` class needs a method that the GUI can call when it's time to redisplay. Let's call this method `reconfigure()`. It will need four arguments:

- A string representing the name of the new source file.
- A boolean that controls whether or not lines should be displayed.
- Colors for keywords.
- Colors for comments.

The method should not paint directly to the screen, because painting is always relegated to the `paint()` method. Our `reconfigure()` method will simply record its four arguments and then call `repaint()`. This will trigger a behind-the-scenes chain of events that will clear the canvas and call `paint()`. When `paint()` runs, it will know what to do (what file to read, what colors to use, whether it should underline), because it will read the values stored by `reconfigure()`.

We can now write the skeleton of `FancySrcCanvas`:

```
public class FancySrcCanvas extends Canvas
{
    private String    fileName;
    private boolean   showLines;
    private Color     keywordColor, commentColor;

    FancySrcCanvas()
    {
        // To do
    }

    void reconfigure(String file, boolean line,
                    Color kColor, Color cColor)
    {
        fileName = file;
        showLines = line;
        keywordColor = kColor;
        commentColor = cColor;
        repaint();
    }

    public void paint(Graphics g)
    {
        // To do
    }
}
```

The body of the constructor and the `paint()` method have been left for later. The constructor will be trivial, but `paint()`, as you

might expect, will be substantial.

Once again, as you saw with the `ColorChoice` class, subclassing allows us to create clean, encapsulated code. If we did not use a canvas subclass, the painting would happen in the `paint()` method of the main frame subclass. This painting code would be jumbled in along with all the other code. With subclassing, we know that all the painting code, and nothing except the painting code, is to be found in `FancySrcCanvas`.

Painting Colored Code

Now we have a workable concept for the `FancySrcCanvas` class, so we can fill in the details. The boring details go in the constructor. The interesting ones go in the `paint()` method.

Let's dispense with the boring details first. We need to choose a font and make some decisions about how to lay out the lines of text. The values we'll use here are somewhat arbitrary. We need to decide on a y coordinate for the topmost line of code. (Remember, when you paint text, you specify the text's baseline, not the top of the text). We need to decide how much vertical space to leave between consecutive lines of code, and we need to choose an x coordinate for the text. [Figure 17.10](#) shows how text will be positioned.

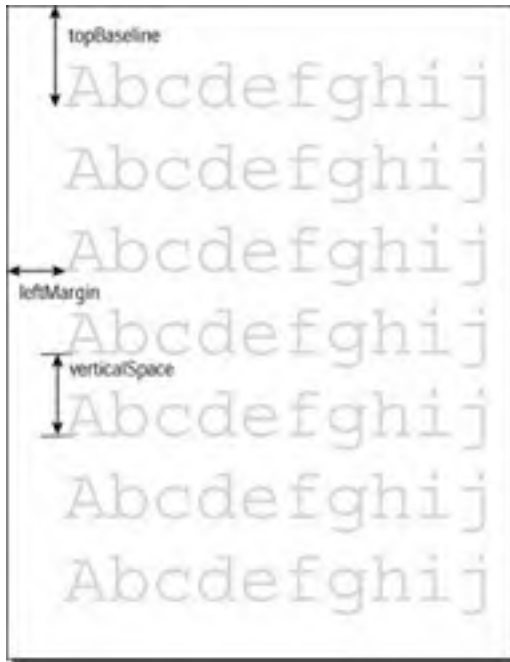


Figure 17.10: Positioning text

We'll use a plain monospaced 16-point font. (Remember, monospaced fonts are always best for displaying source code.) The topmost baseline will be at 20. Every line of text will be 18 pixels below the previous line. The x-coordinate of all text will be 9. These values were arrived at after a fair amount of boring experimentation. The result is the following constructor for

`FancySrcCanvas`:

```
FancySrcCanvas ()
{
    font = new Font("Monospaced", Font.PLAIN, 16);
    topBaseline = 20;
    leftMargin = 9;
    verticalSpace = 18;
}
```

With the font we have chosen, each character is 10 pixels wide. This number will be extremely useful later on. For now, can you guess why it's important?

There is a riddle that brings a knowing gleam to the eyes of experienced programmers, even though it isn't very funny. *How do you fit five elephants into a Volkswagen Beetle? Answer: two in the front, three in the back.* It isn't a good riddle, but it's a good example of top-down development, where you begin with an overall design idea, breaking each piece down into successively more refined designs until there is nothing left to do but implement your solution. Of course, if the design doesn't work, it's not the fault of the elephants.

We will take a top-down approach to developing the `paint()` method of `FancySrcCanvas`, starting with what we know. We know that the horizontal lines must be painted if `showLines` is `true`. We also know that the text must be painted if a source file has been specified. That is, if `fileName`, which is initialized to `null`, is no longer `null`. That's all we know, but it's enough to start. Here's our `paint()` method:

```
public void paint(Graphics g)
{
    if (showLines)
        paintLines(g);
    if (fileName != null)
        paintText(g);
}
```

Now we have to create the `paintLines()` and `paintText()` methods. `paintLines()` seems easy. Starting from the topmost baseline, horizontal lines must be drawn across the entire width of the canvas. Lines must be drawn `verticalSpace` pixels apart, down to the bottom of the canvas. It would all be simple, if only we knew how wide and tall the canvas is.

Fortunately, Canvas has a `getSize()` method that returns an instance of `java.awt.Dimension`. This very simple class has variables `width` and `height`. So the code can use `getSize().width` and `getSize().height` to determine the size of the canvas.

Here is the `paintLines()` code:

```
private void paintLines(Graphics g)
{
    g.setColor(Color.lightGray);
    int height = getSize().height;
    int width = getSize().width;
    for (int y=topBaseline; y<height; y+=verticalSpace)
        g.drawLine(0, y, width, y);
}
```

Now it's time to write `paintText()`. We don't yet know how we're going to draw text in three colors, but we don't have to know. This is top-down development. Let's stay with what we *do* know. There's a Java source file whose name is found in the variable `fileName`. We know that `paintText()` will need to read each line in turn from that file and paint the line. So here is the method, with the issue of painting multicolored text deferred for later consideration:

```
1. private void paintText(Graphics g)
2. {
3.     g.setFont(font);
4.
5.     try
6.     {
7.         // Create the readers.
8.         FileReader fr = new FileReader(fileName);
9.         LineNumberReader lnr = new LineNumberReader(fr);
10.
11.        // Read & display.
12.        String s = "xx"; // Anything but null
13.        int y = topBaseline;
14.        while (s != null)
15.        {
16.            s = lnr.readLine();
17.            if (s == null)
18.                break;
19.            paintOneSourceLine(g, s, y);
20.            y += verticalSpace;
21.        }
22.
23.        // Close the readers.
24.        lnr.close();
25.        fr.close();
26.    }
27.
28.    catch (IOException x)
29.    {
30.        System.out.println("Trouble!" + x.getMessage());
31.    }
32. }
```

The method uses a file reader chained to a line number reader. You were introduced to readers in [Chapter 13, "File Input and Output"](#). The `while` loop in lines 14-21 reads lines of text from the file until the `readLine()` call on line 16 returns `null`, indicating that the end of the file has been reached. (On line 12, `s` has to be initialized to any non-`null` string, so that the loop won't terminate the first time through.)

The variable `y` determines the baseline of the next line of text to be painted. On line 13, `y` is initialized to `topBaseline`. Every pass through the loop, it is incremented by `verticalSpace` (line 20).

Text is painted on line 19, where a call is made to `paintOneSourceLine()`. We'll write this method shortly. Its arguments are the `Graphics` object, the string to be painted (`s`), and the `y`-coordinate of the text (`y`).

We have deferred thinking about how to paint multicolored source text until we could create a good structure for the `paint()` method of `FancySrcCanvas`. That structure is now in place, so it's time to decide how to paint the code. Here's the skeleton of `paintOneSourceLine()`:

```
private void paintOneSourceLine(Graphics g,
                               String srcLine,
                               int y)
{
    ...
}
```

Here's the strategy. First, the method will paint the entire text line in black, whether or not it contains any keywords or comments. Then the line will be inspected to see if it contains any keywords or comments. If so, part of the text will be painted again, in the appropriate color. As you will see, there are methods in the `String` class that make this easy.

Let's start by painting the entire line in black. We don't have to call `setFont()` because that call was made already, in `paintText()`:

```
// First paint entire line in black.
g.setColor(Color.black);
g.drawString(srcLine, leftMargin, y);
```

That was easy. Now to detect and render comments. Comments begin with a double slash (//) and continue through the end of the line. So the code needs to answer the following questions:

Does the string contain a double slash?

If so, where is the double slash?

If the answer to the first question is "no," there is no comment to paint.

Fortunately, class `String` has a method called `indexOf()`. Its argument is another string. If the argument string appears anywhere in the executing object string, the method returns the position of the argument string within the executing object string. For example, if `s1` is `whether` and `s2` is `the`, `s1.indexOf(s2)` is 3. If the argument string does not appear in the executing object string, `indexOf()` returns -1. For example, if `s1` is `whether` and `s2` is `heather`, `s1.indexOf(s2)` is -1.

The comment-painting code uses another method of the `String` class: `substring()`. You were introduced to this method in [Chapter 12, "The Core Java Packages and Classes."](#) When called with a single `int` argument, it returns the portion of the string beginning at the argument position. For example, if `s1` is `whether`, `s1.substring(2)` is `ether`.

Here is the code that paints comments:

```
1. // Paint comment (if any).
2. int commentIndex = srcLine.indexOf("//");
3. if (commentIndex >= 0)
4. {
5.     g.setColor(commentColor);
6.     String comment = srcLine.substring(commentIndex);
7.     int x = charIndexToX(commentIndex);
8.     g.drawString(comment, x, y);
9. }
```

To illustrate how this code works, consider what happens when `srcLine` is
`height += 25; // Increment height`

The comment begins at character position 14, so `commentIndex` is 14. On line 6, `comment` is `//increment height`. This is the string that is overpainted in the comment color, at line 8.

Line 7 makes a call to `charIndexToX()`, which returns the x-coordinate where the comment will be painted. This value must be calculated exactly, so that the new text will exactly overwrite the black text. This method is

```
private int charIndexToX(int charIndex)
{
    return leftMargin + 10*charIndex;
}
```

Earlier in this section, you read that in a 16-point monospaced font, each char is 10 pixels wide. This implies that, for example, the 18th character in any line is 180 pixels to the right of the 0th character. And the 0th character is always painted at `leftMargin`. So the x-coordinate of the `n`th character is `leftMargin + 10*n`. This is the formula used by `charIndexToX()`.

So far our `paintOneSourceLine()` code is

```
private void paintOneSourceLine(Graphics g,
                               String srcLine,
                               int y)
{
    // First paint entire line in black.
    g.setColor(Color.black);
    g.drawString(srcLine, leftMargin, y);

    // Paint comment (if any).
    int commentIndex = srcLine.indexOf("//");
    if (commentIndex >= 0)
    {
        g.setColor(commentColor);
        String comment = srcLine.substring(commentIndex);
        int x = charIndexToX(commentIndex);
        g.drawString(comment, x, y);
    }

    ...
}
```

We are ready to deal with keywords, but we have to be careful. If we just search for keywords and overwrite them in the right color, we could get confounded by a line like this:

```
x = 16; // try to while away the time
```

The code contains no Java keywords, but the comment does. When the line is being searched for keywords, the search should not include the comment. This will not guarantee that the code will never erroneously color non-keyword text, but it guards against one common situation. (Making the keyword search 100% foolproof would be a daunting task. It would overwhelm the code and would seriously reduce the learning value of the project. Not searching comments will be enough for our purposes. Exercise 5 at the end of this chapter invites you to think more about the problem.)

If the software is going to search for keywords, it needs to know which strings are keywords. The `FancySrcCanvas` class needs an array of strings that are Java keywords. Here it is:

```
private String[] keywords =
{
    "abstract", "boolean", "break", "byte", "case", "catch",
    "char", "class", "continue", "default", "double", "do",
    "else", "extends", "false", "final", "float", "for",
    "if", "implements", "import", "instanceof", "int",
    "interface", "long", "new", "null", "package", "private",
    "protected", "public", "return", "short", "static",
    "super", "switch", "this", "throws", "throw", "true",
    "try", "void", "while"
};
```

Actually, the list is incomplete. It only includes Java keywords that were introduced in this book. There are a handful of others. Strictly speaking, `null`, `true`, and `false` are not keywords, but something similar.

Here is the skeleton of the remainder of the `paintOneSourceLine()` code:

```
// Search every position in string, through comment,
// for any keyword.
g.setColor(keywordColor);
int lastCharPosition = srcLine.length()-1;
if (commentIndex >= 0)
    lastCharPosition = commentIndex - 1;
for (int index=0; index<=lastCharPosition; index++)
{
    ...
}
```

The `for` loop will search every position in the line of code, through `lastCharPosition`, to see if it begins with any entry in the `keywords` array. If the line does not contain a double-slash comment, `lastCharPosition` is set to the last character position in the line. If a double-slash comment is present, `lastCharPosition` is set to the last character position before the comment. For example, suppose the source line is

```
x = new Line();// Construct a line
```

The `for` loop will check each of the following substrings:

```
x = new Line();
 = new Line();
= new Line();
 new Line();
new Line();
ew Line();
w Line();
 Line();
Line();
ine();
ne();
e();
();
);
;
```

Each of the substrings will be compared against each entry in the `keywords` array. The code will use two methods of `String` that were presented in [Chapter 12](#), `substring()` and `startsWith()`. The `substring()` method takes an `int` argument. It returns the portion of the original string beginning at the specified index. For example, if `s1` is `Meryl Streep`, `s1.substring(9)` is `eep`. The `startsWith()` method takes a string argument. It returns `true` if the original string starts with the argument string. For example, if `s1` is `Meryl Streep` and `s2` is `Me`, `s1.startsWith(s2)` is `true`.

Now the body of the `for` loop can be filled in:

```
1. for (int index=0; index<=lastCharPosition; index++)
2. {
3.     // Search at this position for every keyword.
4.     String sub = srcLine.substring(index);
5.     for (int i=0; i<keywords.length; i++)
6.     {
7.         if (sub.startsWith(keywords[i]))
8.         {
9.             int x = charIndexToX(index);
10.            g.drawString(keywords[i], x, y);
11.            break; // Can't be any more keywords here
12.        }
13.    }
14. }
```

Recall that the `Graphics` object has already had its color set to the keyword color. Line 9 makes use of the `charIndexToX()` method, which was written for the comment-painting code, to compute where to overdraw the keyword string.

That's all for the `FancySrcCanvas` class. Here is the whole class listing, all in one place:

```
package fancysrc;

import java.io.*;
import java.awt.*;

class FancySrcCanvas extends Canvas
{
    private String[] keywords =
    {
        "abstract", "boolean", "break", "byte", "case", "catch",
        "char", "class", "continue", "default", "do", "double",
        "else", "extends", "false", "final", "float", "for",
        "if", "implements", "import", "instanceof", "int",
        "interface", "long", "new", "null", "package", "private",
        "protected", "public", "return", "short", "static",
        "super", "switch", "this", "throws", "throw", "true",
        "try", "void", "while"
    };

    private Font          font;
    private int           topBaseline;
    private int           leftMargin;
    private int           verticalSpace;
    private String        fileName;
    private boolean       showLines;
    private Color         keywordColor, commentColor;

    FancySrcCanvas()
    {
        font = new Font("Monospaced", Font.PLAIN, 16);
        topBaseline = 20;
        leftMargin = 9;
        verticalSpace = 18;
    }

    void reconfigure(String file, boolean line,
                     Color kColor, Color cColor)
    {
        fileName = file;
        showLines = line;
        keywordColor = kColor;
        commentColor = cColor;
        repaint();
    }

    public void paint(Graphics g)
    {
        if (fileName == null)
            return;

        if (showLines)
            paintLines(g);

        paintText(g);
    }

    private void paintLines(Graphics g)
    {
        g.setColor(Color.lightGray);
        int height = getSize().height;
        int width = getSize().width;
        for (int y=topBaseline; y<height; y+=verticalSpace)
            g.drawLine(0, y, width, y);
    }

    private void paintText(Graphics g)
    {
        g.setFont(font);

        try
        {
            // Create the readers.
            FileReader fr = new FileReader(fileName);
            LineNumberReader lnr = new LineNumberReader(fr);

            // Read & display.
            String s = ""; // Anything but null
            int y = topBaseline;
            while (s != null)
            {
                s = lnr.readLine();
                if (s == null)
                    break;
                if (s.startsWith("import ") || s.startsWith("package "))
                    s = keywordColor + s;
                else if (s.startsWith("//"))
                    s = commentColor + s;
                else
                    s = s;
                g.drawString(s, leftMargin, y);
                y += verticalSpace;
            }
        }
        catch (IOException e)
        {
            // Ignore
        }
    }
}
```

```
        break;
        paintOneSourceLine(g, s, y);
        y += verticalSpace;
    }

    // Close the readers.
    lnr.close();
    fr.close();
}

catch (IOException x)
{
    System.out.println("Trouble! " + x.getMessage());
}
}

private void paintOneSourceLine(Graphics g,
                                String srcLine, int y)
{
    // First paint entire line in black.
    g.setColor(Color.black);
    g.drawString(srcLine, leftMargin, y);

    // Paint comment (if any).
    int commentIndex = srcLine.indexOf("//");
    if (commentIndex >= 0)
    {
        g.setColor(commentColor);
        String comment = srcLine.substring(commentIndex);
        int x = charIndexToX(commentIndex);
        g.drawString(comment, x, y);
    }

    // Search every position in string, through comment,
    // for any keyword.
    g.setColor(keywordColor);
    int lastCharPosition = srcLine.length();
    if (commentIndex >= 0)
        lastCharPosition = commentIndex - 1;
    for (int index=0; index<=lastCharPosition; index++)
    {
        // Search at this position for every keyword.
        String sub = srcLine.substring(index);
        for (int i=0; i<keywords.length; i++)
        {
            if (sub.startsWith(keywords[i]))
            {
                int x = charIndexToX(index);
                g.drawString(keywords[i], x, y);
                break; // Can't be any more keywords here
            }
        }
    }
}

private int charIndexToX(int charIndex)
{
    return leftMargin + 10*charIndex;
}
}
```

Putting It All Together

The [previous section](#) presented the entire listing for `FancySrcCanvas`, which is the longest of the project's three source files. You already saw `ColorChoice` in the "Specifying Colors" Section. That leaves only `FancySrcFrame`, which follows. You have seen its important pieces, but they were *in pieces*. Here it is, all in one place:

```
package fancysrc;

import java.awt.*;
import java.awt.event.*;

public class FancySrcFrame extends Frame implements
    ActionListener, ItemListener
{
    private MenuItem      openMI, exitMI;
    private String        fileName;
    private Checkbox      showLinesBox;
    private ColorChoice   keywordChoice, commentChoice;
    private FancySrcCanvas srcCanvas;
    private FileDialog    dialog;

    FancySrcFrame()
    {
        // Build menu.
        MenuBar mbar = new MenuBar();
        Menu fileMenu = new Menu("File");
        openMI = new MenuItem("Open...");
        openMI.addActionListener(this);
        fileMenu.add(openMI);
        exitMI = new MenuItem("Exit");
        exitMI.addActionListener(this);
        fileMenu.add(exitMI);
        mbar.add(fileMenu);
        setMenuBar(mbar);

        // Build control panel.
        Panel panel = new Panel(); // Uses flow layout
        showLinesBox = new Checkbox("Show lines");
        showLinesBox.addItemListener(this);
        panel.add(showLinesBox);
        keywordChoice = new ColorChoice();
        keywordChoice.addItemListener(this);
        keywordChoice.select(1);
        panel.add(new Label("Keyword color"));
        panel.add(keywordChoice);
        commentChoice = new ColorChoice();
        commentChoice.addItemListener(this);
        commentChoice.select(2);
        panel.add(new Label("Comment color"));
        panel.add(commentChoice);
        add(panel, BorderLayout.NORTH);

        // Build text display panel.
        srcCanvas = new FancySrcCanvas();
        add(srcCanvas, BorderLayout.CENTER);

        // Set to a reasonable size.
        setSize(720, 550);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == openMI)
        {
            if (dialog == null)
                dialog = new FileDialog(this, "Source File",
                    FileDialog.LOAD);
            dialog.setVisible(true); // Modal
            if (dialog.getFile() == null)
                return; // Canceled
            fileName = dialog.getDirectory() + dialog.getFile();
            boolean underline = showLinesBox.getState();
            Color keywordColor =
                keywordChoice.getSelectedColor();
            Color commentColor =
```

```
        commentChoice.getSelectedColor();
        srcCanvas.reconfigure(fileName, underline,
            keywordColor, commentColor);
    }

    else // Must be "Exit" menu item
        System.exit(0);
}

// Called if user activity in checkbox or
// either choice.
public void itemStateChanged(ItemEvent e)
{
    boolean underline = showLinesBox.getState();
    Color keywordColor = keywordChoice.getSelectedColor();
    Color commentColor = commentChoice.getSelectedColor();
    srcCanvas.reconfigure(fileName, underline,
        keywordColor, commentColor);
}

public static void main(String[] args)
{
    (new FancySrcFrame()).setVisible(true);
}
}
```

The only part of this code that has not been explained already is the `itemStateChanged()` method. This is called when the user checks the Show lines check box or either color choice. There is no need to call `getSource()` and figure out which component caused the method call, because the response is the same in any case: A call is made to the `reconfigure()` method of the `FancySrcCanvas`.

Note The complete source to this project is on your CD-ROM, in the `FinalProjectSource` directory.

Goodbye! Don't Forget to Write!

Please take a moment to appreciate what happened in this chapter. You observed the development of a multisource application involving several hundred lines of code:

- The program uses `awt` components for input and paints graphics for its output.
- Each of the three source modules defines a subclass.
- One of the classes implements not just one interface, but two.
- There are loops and conditional statements, and an array.
- There are calls to methods that throw exceptions.

And it all made sense. You saw how it fit together. You deserve sincere congratulations for the work you had to do in [Chapters 1 through 16](#). Without that foundation, this chapter would make no sense at all. You are well on your way toward mastery of the Java language.

This book is by no means a complete introduction. There is a lot more to be said about the language, the core classes, and programming techniques. With your strong foundation, you're now qualified to learn it all.

Were the animated illustrations beneficial? As far as I know, and as far as anybody at Sybex knows, this is the first computer book to use them. If you have any suggestions for how they can be improved, or ideas for new ones, please e-mail them to groundupjava@sgsware.com.

Are you interested in learning more about Java? Would you like to see another volume, picking up where this one leaves off, also based on animated illustrations? Please write.

Exercises

Note The solutions to these exercises are in [Appendix B](#).

1. Write a program that creates a frame with a File menu. The menu should have two items, Save... and Exit. When Save... is selected, the code should display a file dialog box, configured for saving a file. When the user has specified a file via the dialog box, your code should output the name of the file. All the information you need is on the API page for `java.awt.FileDialog`.
2. The `FileDialog` class has a `setDirectory()` method that controls which directory the dialog box will display. Look up the method description in the API to become familiar with how it works. Modify the final project code so that when the file dialog box appears, it displays one of the directories on your computer where you have stored some of your own Java source code. This will make it easier to display your own work.
3. Write an application that displays a canvas subclass in a frame, at Center. The frame does not contain any other components.

Use the following code as the `paint()` method for the canvas subclass:

```
1. public void paint(Graphics G)
2. {
3.     g.setFont(new Font("Serif", Font.PLAIN, 24));
4.     g.setColor(Color.blue);
5.     g.drawString("Look at this!", 0, 0);
6. }
```

Run the program. Do you see what you expected to see? How do you explain the results?

Now change line 5 to this:

```
g.drawString("A bluejay in a quagmire", 0, 0);
```

Now do you see what you expected to see? Again, how do you explain the results?

4. The `FancySrcCanvas` class has an array of Java keywords. In that array, `throws` comes before `throw`. Otherwise, the list is alphabetical. Why does `throws` come before `throw`?
5. There are several situations in which the project code would improperly draw text in the keyword color. How many of these situations can you name?
6. How would you modify the project code so that `null`, `true`, and `false` are not rendered in the keyword color?

Appendix A: Downloading and Installing Java

This is the most important part of this book. Java is not a spectator sport. The best way to learn and enjoy it is to use it. Every chapter of this book except [Chapter 1](#) has programming assignments. You can't do them if your computer doesn't have Java. Equally important, the animated illustrations are all Java programs, and you can't run them without Java. So take the time right now to download and install it. You will be richly rewarded for your effort. Java is an educational tool that will keep you fascinated for the rest of your life.

Overview of the Process

We'll start these instructions with a brief overview of what is involved in downloading and installing. Then you should follow whichever one of the brand-specific sections corresponds to your own computer.

You're going to go to a Javasoftware Web page and download two very large files (tens of megabytes). If you have fast Internet access, congratulations. If you don't, each download could take several hours. If that's the case, consider starting one download at the end of the day. The file will be there for you in the morning. That evening, do the same with the second file.

The first file is Java itself. It is an archive containing the Java compiler, the Java Virtual Machine, and various other helpful programs. (If you don't know what a compiler or Java Virtual Machine are, they're discussed in [Chapter 2](#).) The official name for this download is the SDK, or Software Developer's Kit. The second file contains the API pages, a huge collection of HTML pages that describe the core Java packages and classes. You won't need the API pages until [Chapter 12](#), but you might as well download them as soon as possible.

Before you run any Java program (including the compiler, which is itself a Java program), you have to add the location of Java's executables to your `PATH` environment variable. You may also need to set the `CLASSPATH` environment variable. There are many ways to set these values. The approach presented here involves creating a script to be run manually when you are ready to view the animated illustrations or play with Java. Avoid modifying boot-time or login scripts, because a small typing error can get you into a lot of trouble. Also, manual scripts are easier to undo.

The following section is about installing Java in Windows. If you use a Macintosh, please skip to the "[Macintosh](#)" section later in this appendix.

Windows

This section will walk you through downloading and installing Java on Windows-based computers.

You download Java for Windows from <http://java.sun.com/j2se/1.4/download.html>. This page contains a list of platforms for which the current version of Java is available. For most of these platforms, you can download a JRE (Java Runtime Environment) or an SDK (Software Developer's Kit). *You want the SDK, not the JRE.* At or near the top of the list, you will find Windows (U.S. English only) and Windows (all languages, including English). Click on the link for Windows (all languages, including English).

This opens a page for specifying optional personal information. Then you move on to a license agreement page. If you do not accept the license, you will not be allowed to proceed. The next page allows you to download the file you want. The file has a complicated name, something like `j2sdk-1_4_1_02-windows-i586.exe`.

Click on the link to start the download. If your browser asks you to choose between running the program from its current location or saving it to disk, save it to disk. You can save it anywhere you like, and you can delete it after you run it. You might as well save it in the `\` directory of your C disk.

Execute the file by clicking on its icon in Windows Explorer. The installation wizard begins by asking you to accept the license policy (again). Then you're asked where you want the Java files to be placed. Put them in the `\` directory of your C disk. We'll assume that you use the default. The default name is something complicated, like `j2sdk1.4.1_02`.

Next, you're asked to choose which parts of installation you want. You must select the program files. You don't need any of the other parts, but if you have the disk space, you might be interested in the demos.

Then the installation wizard finishes its work. When it's done, your disk contains a new structure that looks something like [Figure A.1](#).

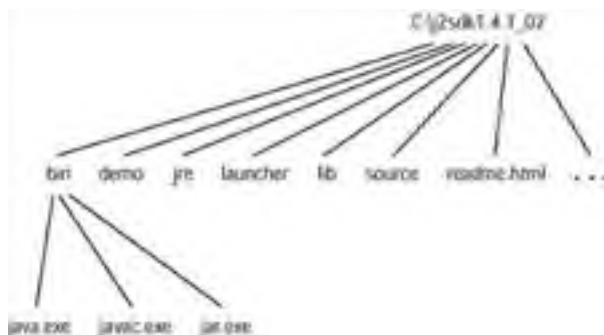


Figure A.1: Windows SDK file layout

The figure shows the three most important files in the `bin` subdirectory: the Java Virtual Machine (`java.exe`), the Java compiler (`javac.exe`), and the `jar` archive tool (`jar.exe`). Throughout this book, you will need the Java Virtual Machine and the compiler. You may need the archive tool shortly.

All the Java files are in place. Now create a directory where you will write Java programs. Again, you can call it whatever you want and put it wherever you like, but simpler is better. Here we'll assume your programming directory is called `C:\MyJavaCode`.

The next step is to create a batch file for setting your `PATH` and `CLASSPATH` environment variables. The `PATH` variable tells the operating system where to look for executable files when you run a program from the command line of a Command Prompt window. (And that is how you will run the animated illustrations and compile and execute your own programs.) The `CLASSPATH` variable tells Java where to look for class files. (That won't make sense unless you've read [Chapter 8](#).) You can call your script anything you like, and you can store it anywhere you like. The whole point of a batch file is to simplify things and reduce typing, so call it something simple and save it somewhere easy to remember. We will call it `\ja.bat`. It looks like this:

```
SET PATH=%PATH%;C:\j2sdk1.4.1_02\bin
SET CLASSPATH=.;D:\AnimatedIllustrations
CD C:\MyJavaCode
```

You don't have to use exactly this script, but if you like it there's a copy on the CD-ROM, in the `ExampleScripts\Windows` subdirectory. If you want to use it as-is, just copy it to the root directory of your primary disk drive. You might want to copy it anyway and use a text editor such as `Notepad` to edit it, rather than typing in your version from scratch.

The first line of the script appends Java's `bin` directory to your `PATH` environment variable, so that you will be able to run programs like `java`, `javac`, and `jar`. Remember that you will be running them by typing command lines into a Command Prompt window, not by double-clicking on icons.

The second line sets the `CLASSPATH` environment variable to the current directory, plus the directory on this book's CD-ROM where the animated illustration programs are stored. (If your CD-ROM drive letter is something other than `D`, substitute the appropriate letter. Alternately, you can copy the `AnimatedIllustrations` directory to your hard drive. That way you don't have to make sure the CD-ROM is loaded whenever you want to run an animated illustration. If you copy the `AnimatedIllustrations` directory to your hard drive, replace `D:\ AnimatedIllustrations` in the script with the full path, including drive letter, of the copied version of the `AnimatedIllustrations` directory.)

The third line of the script takes you into the directory where you will create and store the Java programs that you will write, in answer to some of the questions at the end of each chapter.

To test your work, open a Command Prompt window. Run your batch file script by typing `\ja`. The script runs. Note that it only affects the one Command Prompt window you are working in. If you close that window, you will have to open another one and then run the batch file again.

To make sure your script ran properly, type `java -version`. You get a message that tells you which version of Java is running, along with some other obscure, cryptic information that probably means something important to someone. This message means that you have installed Java and your `PATH` variable is set correctly. If the command doesn't work, make sure the full pathname of the `bin` directory in your script is spelled correctly, and that the directory contains `java.exe`.

Now type `java welcome.Welcome`. You should see a simple welcoming screen. This means that your `CLASSPATH` variable is set correctly. If you don't see the welcoming screen, make sure the value assigned to `CLASSPATH` in your script is spelled correctly, and that the directory is the `AnimatedIllustrations` directory from the CD-ROM. If you are running the animated illustrations directly from the CD-ROM (that is, if you didn't copy the files to your hard drive), make sure the CD-ROM is in the correct drive.

That takes care of the Java program files. They are all you need until [Chapter 12](#), where you will also need the API pages. These are a huge number of HTML pages to be viewed with the Web browser of your choice. You download them as a single zip file that you will have to extract.

You begin the download at the same page you visited before: <http://java.sun.com/j2se/1.4/download.html>. This page contains a list of several dozen products that you can download. Near the end of the list, you see J2SE 1.4.1 Documentation. Click on this item's Download link. After accepting another license agreement, you see a page with a link for downloading `j2sdk-1_4_1-doc.zip`. Click on the link. You're prompted to specify where you want to put the zip file. Put it in the directory where you stored your Java files. We recommend `C:\j2sdk1.4.1_02`.

Now you might have to extract the zip file. Some versions of Windows present a zip file as if it were a directory, extracting files only as needed. This saves a lot of space. If your system does this for you and it's satisfactory, you're done. To find out, open a Windows Explorer window and have it display your `C:\j2sdk1.4.1_02` directory. If `j2sdk-1_4_1-doc.zip` looks like a directory rather than a single file, you don't need to extract it if you don't want to. Otherwise, you need to extract.

To extract, you could double-click on the `j2sdk-1_4_1-doc.zip` icon and use Winzip to unpack the archive. But you might not have Winzip. Besides, there is a slicker way. If you haven't done so already, run your batch file script by typing `\ja`. Next, type `cd` into the directory that contains the `j2sdk-1_4_1-doc.zip` file. Unless you have done your own thing, the command to do so is `cd C:\j2sdk1.4.1_02`

Now type the following command:

```
jar xvf j2sdk-1_4_1-doc.zip
```

The `jar` command is one of the useful Java executables. It became usable when you ran the script and added `C:\j2sdk1.4.1_02\bin` to your path. Jar stands for *Java Archive*. It is like Winzip, but you run it from the command line. Fortunately, the `jar` file format is compatible with the `.zip` format, so you can use `jar` to extract any `.zip` archive.

Creating Program Files

Congratulations! You are ready to go. Right now you can run any animated illustration in the book. And please do... they are an essential part of your *Ground-Up Java* experience.

The other essential part of your experience is writing your own Java programs. You will do this when you work on the "write-a-program" questions at the end of the chapters, and of course you can write programs that implement your own ideas. We have already recommended that you do this in a directory called `\MyJavaCode`, and have your script "cd" in that directory. But now the question is, how do you create Java code? [Chapter 2](#) explained that writing a Java program means creating one or more files, called source files, in plain text format, with names that end with `.java`.

There are two ways to create Java source files:

- Use a general-purpose editor.
- Use an Integrated Development Environment (IDE).

You have several general-purpose editors installed on your system, including Notepad and Wordpad. It's a good idea to start with one of these; they are good enough for small programs. Use a fixed-width font like Courier to make your code line up nicely.

As you will learn from experience, you don't just write a program. The development process is an ongoing cycle of writing, testing, and modifying. So when you think you have finished writing your program, don't close your editing window. Leave it around, because in all likelihood you will want to make modifications or fix bugs. (Yes, this could happen even to you.)

After a while, you might get a vague sense that life could be better somehow. You might be ready for an IDE, or Integrated Development Environment. IDEs are products that help you create, maintain, debug, and keep track of Java programs. Many common operations (such as compiling) are achieved with a single button click. There are lots of IDEs on the market, ranging in price from free to expensive. It would be inappropriate to recommend one here, but if you type **Java + IDE** into your favorite Web search engine, you will get plenty of information.

A good IDE is a good thing, and a great IDE will greatly enhance your productivity. But a word of warning: Your goal right now is *not* to create large Java programs efficiently. Your goal is to learn as much as possible about Java. IDEs shield you from repetitive tasks. Before you start using them, it's a good idea to spend some time learning all the ins and outs and

details of Java, so that you'll know what the IDE is shielding you *from*. Spend some time with a general-purpose editor before you move on to an IDE.

The rest of this appendix is about installing Java on Macintosh. If you only have a Windows PC, you can skip the rest. Have fun!

Macintosh

This section will walk you through downloading and installing Java on Macintosh-based computers.

You download Java for Macintosh directly from Apple Computers. You will need three separate pieces:

- Mac OSX Developer Tools
- Java 1.4.1 Developer Tools Update
- Java 1.4.1. for Mac OS X and QTJava

First you will need to register as an Apple developer. This is free (as are the downloads), but you must do it before you can access the download sites. Just type this into your browser:

```
http://connect.apple.com
```

This will take you to the Apple Developer Connection site. From here, you can log in and download the software you want. If you are not yet a member, you must become one. Click the Join ADC button on the left side of the page, and answer the questions on the form. You're granted a user name and password to log in to the download site.

Once you've obtained your membership, go ahead and log in (same site as above). On the resulting page, click the Download Software link. You see a list of software packages along the left side of the page. Click on the Mac OS X link, and you see a list of possible items for download. Click the Download button immediately to the right of Dec 2002 Mac OS X Developer Tools. Downloading commences immediately. This file, `Dec2002DevToolsCD.dmg`, is 301.2MB and takes a little over an hour to download over a DSL line. Make a note of where you store the file on your local hard drive.

Now you need to update your Developer Tools. Use the Back button on your browser to return to the list of downloadable items. This time, choose Java. Click the Download button immediately to the right of Java 1.4.1 Developer Tools Update. Downloading commences immediately. This file, `Java141Developer.dmg`, is 48.6MB and takes about 20 minutes to download over a DSL line. Again, make a note of where you store the file on your local hard drive.

Lastly, you need to get the most recent version of Java for the Mac (1.4.1). Use the Back button to return to the list of downloadable items. Choose Java again. Click the Download button immediately to the right of Java 1.4.1 Update DP102. Downloading commences immediately. This file, `Java141Update1DP102.dmg`, is 37.4MB and takes about 15 minutes to download over a DSL line. Make a note of where you store the file on your local hard drive.

Now it's time to unpack and install the three files you've downloaded. First, the developer tools. Use Finder to navigate to the folder where you've placed the downloads. Locate the file `Dec2002DevToolsCD.dmg`, and double-click on its icon. You still need to be connected to the network during this process, because the `.dmg` file will attempt to mount the disk image of the Developer's Tools package for subsequent installation. You also need administrator-level permission to complete the installation. If all goes well, a small window labeled December 2002 Dev Tools appears. Double-click on the Developer icon and proceed with the installation as directed. When the installer asks you for a destination, select the default, Normal. When the installation has finished, you must reboot your computer before proceeding to the next installation.

Next it's time to unpack and install the Java update. Use Finder to navigate to the folder where you've placed the downloads. Locate the file `Java141Update1DP102.dmg` and double-click on its icon. A small window labeled Java 1.4.1 Update 1 appears. Double-click on the `Java1.4.1Update1.pkg` icon and proceed with the installation as directed. When the installer asks you for a destination, select the default, Normal. When the update has been unpacked and installed, you're directed to restart your computer.

Now it's time to unpack and install the Developer Tools update. Use Finder to navigate to the folder where you've placed the downloads. Locate the file `Java141Developer.dmg` and double-click on its icon. A small window labeled Java 1.4.1 Developer Update appears. Double-click on the `Java1.4.1Developer.mpkg` icon and proceed with the installation as directed. When the installer asks you for a destination, select the default, Normal.

Now all the Java files are in place. Create a directory where you will write Java programs. You can call it whatever you want and put it wherever you like, but simpler is better. Here we'll assume your programming directory is called `~/MyJavaCode`.

The next step is to add two lines to your `.login` or `.cshrc` file for setting your `path`:

```
set path=(lib:/usr/local/bin:/usr/ucb:/bin:/sbin:/usr/bin:/usr/sbin:/usr/etc:/Developer/Tools)
setenv CLASSPATH ./AnimatedIllustrations
alias gotoJava 'cd ~/MyJavaCode'
```

The first line tells the operating system where to look for executable files when you run a program from the command line of a terminal window. That's how you will run the animated illustrations, and compile and execute your own programs.

The second line sets the `CLASSPATH` environment variable to the current directory, plus the directory on this book's CD-ROM where the animated illustration programs are stored. The script assumes you have copied the entire CD-ROM to a directory on your hard drive called `/AnimatedIllustrations`. Doing so will make life simpler, because you can run the illustrations without the CD-ROM. If you don't want to copy the CD-ROM to your hard drive, or if you want to copy it to a different directory, modify the script accordingly.

The third line of the script is an alias that will take you to your Java work directory.

To test your work, log out and then log in again so that the script will execute. Open a terminal window (Applications/Utilities/Terminal in the Finder). To make sure your script ran properly, type `java -version`. You get a message that tells you which version of Java is running, along with some other obscure, cryptic information that probably means something important to someone. If you get this message, you have installed Java and your `PATH` variable is set correctly. If the command doesn't work, make sure the full pathname of the `bin` directory in your script is spelled correctly, and that the directory contains `java`.

Now type `java welcome.Welcome`. You should see a simple welcoming screen. This means that your `CLASSPATH` variable is set correctly. If you don't see the welcoming screen, make sure the value assigned to `CLASSPATH` in your script is spelled correctly, and that the directory is the `AnimatedIllustrations` directory from the CD-ROM. If you are running the animated illustrations directly from the CD-ROM (that is, if you didn't copy the files to your hard drive), make sure the CD-ROM is in the correct drive.

That takes care of the Java program files. They are all you need until [Chapter 12](#), where you will also need the API pages. These are a huge number of HTML pages to be viewed with the Web browser of your choice. You download them as a single zip file that you will have to extract.

You begin the download at <http://java.sun.com/j2se/1.4.1/download.html>. The page contains a list of several dozen products that you can download. Near the end of the list is J2SE 1.4.1 Documentation. Click on this item's Download link. After accepting another license agreement, you come to a page with a link for downloading `j2sdk-1_4_1-doc.zip`. Click on the link. You are prompted to specify where you want to put the zip file. Put it in a directory where you can find it easily. We recommend `~/MyJavaFiles`.

To extract, use `jar` to unzip the documents in the `j2sdk-1_4_1-doc.zip` file. If you haven't done so already, run your batch file script by typing `gotoJava`. Unless you have done your own thing, the command to unzip the file is

```
jar xvf j2sdk-1_4_1-doc.zip
```

The `jar` command is one of the useful Java executables. It became usable when you ran the script and added `C:\j2sdk1.4.1_02\bin` to your path. `jar` stands for *Java Archive*. It is like `Stuffit-Expander`, but you run it from the command line. Fortunately, the `jar` file format is compatible with the `.zip` format, so you can use `jar` to extract any `.zip` archive.

Creating Program Files

Congratulations! You are ready to go. Right now you can run any animated illustration in the book. And please do... they are an essential part of your [Ground-Up Java](#) experience.

The other essential part of your experience is writing your own Java programs. You will do this when you work on the "write-a-program" questions at the end of the chapters, and of course you can write programs that implement your own ideas. We have already recommended that you do this in a directory called `\MyJavaCode`, and have your script "cd" in that directory. But now the question is, how do you create Java code? You will see in [Chapter 2](#) that writing a Java program means creating one or more files, called source files, in plain text format, with names that end with `.java`.

There are two ways to create Java source files:

- Use a general-purpose editor.
- Use an Integrated Development Environment.

You have several general-purpose editors installed on your system, including Notepad and Wordpad. It's a good idea to start with one of these. They are good enough for small programs. Use a fixed-width font like Courier to make your code line up nicely.

As you will learn from experience, you don't just write a program. The development process is an ongoing cycle of writing, testing, and modifying. So when you think you have finished writing your program, don't close your editing window. Leave it around, because in all likelihood you will want to make modifications or fix bugs. (Yes, this could even happen to you.)

After a while, you might get a vague sense that life could be better somehow. You might be ready for an IDE, or Integrated Development Environment. IDEs are products that help you create, maintain, debug, and keep track of Java programs. Many common operations (such as compiling) are achieved with a single button click. There are lots of IDEs on the market, ranging in price from free to expensive. It would be inappropriate to recommend one here, but if you type "Java + IDE" into your favorite Web search engine, you will get plenty of information.

A good IDE is a good thing, and a great IDE will greatly enhance your productivity. But a word of warning: Your goal right now is *not* to create large Java programs efficiently. Your goal is to learn as much as possible about Java. IDEs shield you from repetitive tasks. Before you use them, it's a good idea to spend some time learning all the ins and outs and details of Java, so that you'll know what the IDE is shielding you *from*. Spend some time with a general-purpose editor before you move on to an IDE.

Appendix B: Solutions to the Exercises

Chapter 1

Exercise 1 A cluster of eight bytes can take on approximately 20 quintillion different values. (One quintillion is a 1 followed by 18 zeroes, or 10 to the 18th power.) Estimate the number of different values that a cluster of 16 bytes can have. Just estimate, do not count. Can you think of anything that comes in such quantities?

Solution 1 The exact number of values is 2 to the power of the number of bits. This is 2^{128} , or about 3.4×10^{38} . We can make a good estimate by just squaring the number of possibilities for eight bytes, which is given as approximately 20×10^{18} . The square of that is approximately 400×10^{36} , or approximately 4×10^{38} .

To put this in perspective, there are about 2×10^{11} stars in a typical galaxy, and there are about 10^{10} galaxies in the universe. So 16 bytes can easily store the number of stars in the universe (2×10^{21}).

Exercise 2 The SimCom animated illustration is written in Java. When you run the program, how many virtual machines are at work?

Solution 2 SimCom is a virtual machine that runs on the Java Virtual Machine that runs on your physical computer. So there are two virtual machines.

Exercise 3 Write a SimCom program that adds 255 to the value in byte 31 and stores the result in byte 30. Observe the program's behavior. What do you notice?

Solution 3 The following program adds 255 to the contents of byte 31, and stores the result in byte 30. The program appears in the solutions on the CD-ROM, in answers/ Ch1/Add255.simcom. In addition to the following code, the program also stores the number 1 in byte 29:

```
LOAD 31
ADD 29
STORE 30
HALT
```

SimCom acts as if adding 255 were the same as subtracting 1. We will look at this in more detail in the [next chapter](#).

Exercise 4 Write a SimCom program that computes the square of the value in byte 31 and stores the result in byte 30. What happens when you try to compute the square of 254?

Solution 4 The following program squares the contents of byte 31, and stores the result in byte 30. The program appears in the solutions on the CD-ROM, in answers/Ch1/ Square.simcom:

```
LOAD 31
STORE 29
LOAD 31
ADD 30
STORE 30
LOAD 29
SUB 28
STORE 29
JUMPZ 10
JUMP 2
HALT
```

This program is almost the same as the Times5 program that you saw earlier in [Chapter 1](#). The difference is that instead of using a hard-coded value as the loop counter, the first two lines of this program store the value to be squared in the loop counter.

This program produces 4 as the square of 254.

Exercise 5 What features could be added to SimCom to make it more useful?

Solution 5 This is a subjective issue. It would be reasonable to want more opcodes, especially for multiplying and dividing. More memory would also be good. But bear in mind that, since SimCom forces you to be aware of all features of the architecture, adding more features would just give you more to juggle. The benefit of a high-level programming language such as Java is that you can take advantage of a computer's features without having to think on too low a level.

Chapter 2

Exercise 1 According to [Table 2.1](#), the maximum values for the byte and short data types are 127 and 32767, respectively. Use the Twos-Complement Lab animated illustration to verify this. Which byte and short bit patterns produce the maximum values? In general, which bit pattern produces the maximum value for a two's complement number of N bits?

Solution 1 The maximum-value byte is 01111111. The maximum-value short is 0111111111111111. The general formula is a leading 0 followed by all 1s.

Exercise 2 According to [Table 2.1](#), the minimum values for the byte and short data types are -128 and -32768, respectively. Use the Twos-Complement Lab animated illustration to verify this. What byte and short bit patterns produce the minimum values? In general, what bit pattern produces the minimum value for a two's complement number of N bits?

Solution 2 The minimum-value byte is 10000000. The minimum-value short is 1000000000000000. The general formula is a leading 1 followed by all 0s.

Exercise 3 Launch the Twos-Complement Lab animated illustration by typing `java TwosCompLab`, set the data type to int, and set all the bits to 1. Then set the three bits on the right to 0. Compute the value. Do the same for the byte and short data types. What do you observe?

Solution 3 In each case, the result is -8.

Exercise 4 Launch the Floating-Point Lab animated illustration by typing `java floating.FloatFrame`. Set the rightmost bit to 1 and all other bits to 0. The value represented is 1.4E-45. Try changing various bits' values by clicking on them. Can you create a value that is smaller than 1.4E-45 but still greater than 0?

Solution 4 1.4E-45 is the smallest possible greater-than-zero float value. [Table 2.2](#) says so. Changing any bits in the exponent part yields a bigger power of 2. Changing any bits in the fraction part yields a bigger fraction, unless you set all the fraction bits to 0, which represents an overall value of 0.

Exercise 5 Write a Java application that declares and assigns values to three int variables named x, y, and z. Print out all three values, separated by commas, on a single line.

Solution 5 The following application prints out the values, separated by commas:

```
public class Ch2Q5
{
    public static void main(String[] args)
    {
        int x, y, z;
        x = 10;
        y = 20;
        z = 30;
        System.out.println(x + "," + y + "," + z);
    }
}
```

Exercise 6 *White space* means spaces, tabs, and line-break characters. Type in the VerySimple application from [Chapter 2](#) (reproduced below) and experiment with inserting white space. Does anything change during compilation or execution if you insert extra spaces between `public` and `class`? What if you insert a line break between `public` and `class`? Can you find any adjacent words or symbols such that inserting white space between them changes compilation or execution?

```
public class VerySimple
{
    public static void main(String[] args)
    {
        double age;
        age = 123.456;
    }
}
```

Solution 6 White space between words and symbols has no effect on compilation or execution, unless you put the white space inside a literal string. In that case, of course, the literal string will be changed. This means that you can use white space to make your source code as readable as possible. This is discussed further in [Chapter 3](#), in the section "[White Space and Comments](#)."

Chapter 3

Exercise 1 What happens when a comment appears inside a literal string? (Recall from [Chapter 2](#) that a literal string is a run of text enclosed between double quotes.) What would the following line of code do?

```
System.out.println("A /* Did this print? */ Z");
```

Write a program that includes this line. Does the program print the entire literal string, or does it just print "A Z"?

Solution 1 The program prints the entire literal string. A // or /* inside a literal string doesn't signal the start of a comment.

Exercise 2 What is the value of ~100? What is the value of ~~100? First try to figure it out, and then write a program to print out the values. (Hint: You can figure it out without using pen and paper if you remember something that was discussed in [Chapter 2](#).)

Solution 2 ~100 is -101. ~~100 is 99. Recall from [Chapter 2](#) that to generate the negative of an integer type, invert all its bits and then add one. In other words, if you use ~ to invert an integer's bits, you have almost generated its negative. Almost, but not quite: You still have to add 1. So ~ generates the negative of its argument, minus 1.

The following program performs the computations:

```
public class TildeTest
{
    public static void main(String[] args)
    {
        int n = 100;
        int nTilde = ~n;
        System.out.println("~100 = " + nTilde);
        n = -100;
        nTilde = ~n;
        System.out.println("~-100 = " + nTilde);
    }
}
```

Exercise 3 Write a program that prints out the following values:

```
32 << 3
32 >> 3
32 >>> 3
-32 << 3
-32 >> 3
-32 >>> 3
```

Solution 3 The following program performs the required operations:

```
public class Shift32By3
{
    public static void main(String[] args)
    {
        int x = 32 << 3;
        System.out.println("32 << 3 = " + x);
        x = 32 >> 3;
        System.out.println("32 >> 3 = " + x);
        x = 32 >>> 3;
        System.out.println("32 >>> 3 = " + x);
        x = -32 << 3;
        System.out.println("-32 << 3 = " + x);
        x = -32 >> 3;
        System.out.println("-32 >> 3 = " + x);
        x = -32 >>> 3;
        System.out.println("-32 >>> 3 = " + x);
    }
}
```

The output is

```
32 << 3 = 256
32 >> 3 = 4
32 >>> 3 = 4
-32 << 3 = -256
-32 >> 3 = -4
-32 >>> 3 = 536870908
```

Exercise 4 What are the values of the following expressions? First do the computations mentally. Then write a program to verify your answer.

```
false & ((true^(true&(false|(true|false))))^true)
true | (true^false^false^true&(false|(true&true)))
```

Solution 4 The first expression has the form "false & anything", so its value is false. The second expression has the form "true | anything", so its value is true. The following program verifies this:

```
public class AndAnythingOrAnything
{

```

```
public static void main(String[] args)
{
    boolean a = false & ((true^(true&(false!(true|false))))^true);
    boolean b = true | (true^false^false^true&(false!(true&true)));
    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
}
```

Exercise 5 The following expression looks innocent:

```
boolean b = (x == 0) | (10/x > 3);
```

You can assume x is an int. Write a program that prints out the value of this expression for the following values of x : 5, 2, 0. What goes wrong? (You will see a failure message that you might not be familiar with, because we have not introduced it yet. Don't worry—just try to understand the general concept.) How can you make the code more robust by adding a single character to the expression?

Solution 5 The following program does what the question requires:

```
1. public class Chap3Q5
2. {
3.     public static void main(String[] args)
4.     {
5.         int x = 5;
6.         boolean b = (x == 0) | (10/x > 3);
7.         System.out.println("x=" + x + ", b=" + b);
8.         x = 2;
9.         b = (x == 0) | (10/x > 3);
10.        System.out.println("x=" + x + ", b=" + b);
11.        x = 0;
12.        b = (x == 0) | (10/x > 3);
13.        System.out.println("x=" + x + ", b=" + b);
14.    }
15. }
```

The output from line 7 is "x=5, b=false". The output from line 10 is "x=2, b=true". You don't get any output from line 13. Instead, the JVM returns an error message. Your message may vary based on your JVM rev, but probably you saw the following:
java.lang.ArithmeticException: / by zero at Chap3Qs.main(Chap3Qs.java:12)

When a program prints out a message like this that includes the word "Exception", you know that something has gone wrong. Exceptions are Java's mechanism for indicating program trouble or failure. They're covered in [Chapter 11](#), "Exceptions." The stuff in parentheses at the end of the message says that something went wrong at line 12, so execution was abandoned at that point. The message and a glance at line 12 tell us that we have tried to divide 10 by zero. This is an illegal operation, because dividing by zero is undefined.

To fix the program, just change `|` to `||`. At line 12, the " $x == 0$ " comparison will evaluate to true, so the short-circuit operator will skip the illegal remainder of the expression.

The "Short-Circuit Operators" section of [Chapter 3](#) explained that short-circuit operators let you avoid unnecessary execution of time-consuming code. This question shows that you can also use them to avoid unnecessary execution of code that would generate an error.

Exercise 6 The 32-bit float type is wider than the 64-bit long type. How can a 32-bit type be wider than a 64-bit type?

Solution 6 Longs (64 bits) use two's-complement data representation, and floats (32 bits) use floating-point representation. No matter what data representation is used, there are exactly 2^n possible combinations of n bits. The way to think about this problem is to consider the way that represented numbers are distributed. The long type represents 2^{64} values, evenly distributed along the number line. In other words, the distance between any two consecutive numbers represented by a long is exactly 1.

With floats, the 2^{32} values are not evenly distributed. If you draw a dot on the number line for every number represented by a float, you see a dense cluster near zero. The farther you get from zero, the more sparsely the dots appear. Far out near the extreme positive and negative ends of the range, the dots are very rare indeed. To quantify, the smallest-magnitude float that is greater than zero—in other words, the first number to the right of zero on the number line—is 1.4×10^{-45} . However, the difference between the largest float and the next-smallest float is about 2×10^{31} —a truly astronomical number.

So the 32-bit float type achieves a wider range than the 64-bit long type by distributing its represented values more sparsely.

Exercise 7 Write a program that contains the following two lines:

```
byte b = 6;
byte b1 = -b;
```

What happens when you try to compile the program?

Solution 7 The first line (`byte b = 6;`) is legal. The second line (`byte b1 = -b;`) is a problem. The result of the unary `-` operation is of type `int`, and the code tries to assign an `int` to a `byte`. The compilation will fail. The compiler error may vary depending on your compiler rev, but probably you will get a message that says this:

```
...possible loss of precision: int, required: byte...
```

Chapter 4

Exercise 1 Which of the following are legal method names?

- a. \$25
- b. 25\$
- c. abc_
- d. _ABc

Solution 1 A, C, and D are legal. B is illegal because a method name may not begin with a digit.

Exercise 2 Suppose you want to write a method that returns the diameter of a planet, in millimeters. Since it's your program, you can choose any name you like for the method. Rank the following method names, from worst to best. Use your own judgment as to what makes one method name better or worse than another.

- a. getPlanetDiameter
- b. getSize
- c. getPlanetDiameterMm
- d. getIt
- e. getPlanetSize

Solution 2 Good and bad are subjective. However, a method name that eliminates confusion must be considered better than one that creates confusion, or only eliminates a little confusion. Here are the method names, ranked by order of how much information each name conveys:

```
getIt  
getSize  
getPlanetSize  
getPlanetDiameter  
getPlanetDiameterMm
```

`getIt` tells you nothing at all about what the method does. It's unfortunate how many programmers use similar names and create code that is difficult to understand and expand. The other names tell increasingly more about the method's return value. "Diameter" is better than "Size", because `Size` might be diameter or radius or mass. "DiameterMM" tells us not only the quantity but the units.

Of course, there is a limit to how much a method name should say. The goal is not to maximize the information in the name. The goal is to maximize the *usefulness* of the name. A name that is too long to read easily, or hard to distinguish from a similar name, does not contribute. For example, `getPlanetDiameterMMAsMeasuredByHubbleOnApril112003` is too informative, and it's hard to distinguish from `getPlanetDiameterMMAsMeasuredByHubbleOnApril112003`.

Exercise 3 Suppose a method has the following declaration:

```
static int abc(int x, short y)
```

Suppose this method is called as follows:

```
abc(first, second)
```

Which of the following are legal types for the variables `first` and `second`?

- a. int first, int second
- b. short first, short second
- c. byte first, char second
- d. char first, byte second

Solution 3 B and D are legal. A passed argument may be of any type, provided it is the same as, or narrower than, the type declared by the method. The first argument is declared by the method to be an int, so you can pass a byte, short, char, or int. The second argument is declared by the method to be a short, so you can pass a byte or a short.

Exercise 4 Consider the following method declaration:

```
xyz(double d)
```

Which argument types can a caller pass into this method?

Solution 4 A caller can pass any type that is the same as, or shorter than, the declared type. Since the declared type is double, the caller can pass a byte, short, char, int, long, float, or double.

Exercise 5 In [Chapter 4](#), you learned that if method `iAmVoid` is void, you can't say `int z = iAmVoid();` because there is no value to assign to `z`. What happens if you try? Write a program that does this experiment.

Solution 5 The following application tries to assign a void call to an int:

```
public class AssignVoid  
{  
    static void iAmVoid()  
    {
```

```
        System.out.println("Hello");
    }

    public static void main(String[] args)
    {
        int z = iAmVoid();
    }
}
```

The compilation fails with the following message: "Incompatible types; found, void, required:int at line 10..." (Your actual message may vary, depending on where your compiler came from.)

Exercise 6 In [Chapter 4](#), you saw the following method:

```
static void print3x(int x)
{
    x = 3*x;
    System.out.println("3 times x = " + x);
}
```

The following code prints out "Now z is 10", not "Now z is 30", because the method modifies its own private copy of the argument:

```
int z = 10;
print3x(z);
System.out.println("Now z is " + z);
```

Write a program that proves this.

Solution 6 The following code proves that the method modifies its own copy, leaving the caller's copy alone:

```
public class ProveCallByValue
{
    static void print3x(int x)
    {
        x = 3*x;
        System.out.println("3 times x = " + x);
    }

    public static void main(String[] args)
    {
        int z = 10;
        print3x(z);
        System.out.println("Now z is " + z);
    }
}
```

Chapter 5

Exercise 1 Rewrite the following code to maximize readability:

```
switch (x)
{
    case 100:
        System.out.println("x is big");
        break;
    case 101:
        System.out.println("x is big");
        break;
    case 10:
        System.out.println("x is medium");
        break;
    case -1000:
        System.out.println("x is negative");
        break;
}
```

Solution 1 The 100 and 101 cases can be combined, and the cases can be arranged in ascending numerical order, to produce the following:

```
switch (x)
{
    case -1000:
        System.out.println("x is negative");
        break;
    case 10:
        System.out.println("x is medium");
        break;
    case 100:
    case 101:
        System.out.println("x is big");
        break;
}
```

Exercise 2 Rewrite the following code to make it cleaner:

```
boolean flag = false;
switch (a)
{
    case 1:
        x = 1000;
        flag = true;
        break;
    case 30:
        y = 1000;
        flag = true;
        break;
}
if (!flag)
    z = 1000;
```

Solution 2 The flag just indicates that the switch has a case that matches its argument. So the "z = 1000" assignment happens only if there was no case to match the switch argument. We can eliminate the flag and move the "z = 1000" assignment into the switch's default case:

```
switch (a)
{
    case 1:
        x = 1000;
        flag = true;
        break;
    case 30:
        y = 1000;
        flag = true;
        break;
    default:
        z = 1000;
        break;
}
```

Exercise 3 What happens when the following code is executed with val equal to 10? 100? 1,000? First, decide just by looking at the source code. Then write a program to verify your answer.

```
switch (val)
{
    case 10:
        System.out.println("ten");
    case 100:
        System.out.println("hundred");
    default:
        System.out.println("thousand");
}
```

Solution 3 The code doesn't have any break statements, so every case will fall through to the next one. The output for 10 is

```
ten
hundred
thousand
```

The output for 100 is

```
hundred
thousand
```

And the output for 1000, which is handled by the default code, is

```
thousand
```

The following program verifies the results. Note the for loop, with multiple action in the update:

```
public class SwitchTest
{
    public static void main(String[] args)
    {
        int val = 10;
        for (int i=0; i<3; i++, val*=10)
        {
            System.out.println("\nTesting " + val + " ... ");
            switch (val)
            {
                case 10:
                    System.out.println("ten");
                case 100:
                    System.out.println("hundred");
                default:
                    System.out.println("thousand");
            }
        }
    }
}
```

Exercise 4 Run the WhileLab animated illustration by typing `java loops.WhileLab`. Try changing the value in the condition in the third line. What do you notice about the final value of `a`?

Solution 4 The final values of `a` are always square numbers.

Exercise 5 The description of WhileLab suggests three exercises, which are repeated here. For each desired result, configure the inputs of WhileLab to produce that result. Then verify your work (and make sure WhileLab is trustworthy) by writing an application that duplicates each while loop. The loops should generate the following results:

- The sum of the numbers 1 through 500, inclusive.
- The sum of the even numbers from 50 through 60, inclusive.
- The product of the first 5 odd numbers.

Solution 5 The sum of 1 through 500:

```
int a = 0;
int b = 1;
while (b <= 500)
{
    a = a+b;
    b = b+1;
}
```

The sum of the even numbers from 50 through 60, inclusive:

```
int a = 0;
int b = 50;
while (b <= 60)
{
    a = a+b;
    b = b+2;
}
```

The product of the first 5 odd numbers (the n^{th} odd number is $2n+1$):

```
int a = 1;
int b = 0;
while (b < 5)
{
    a = a * (2*b+1);
    b = b+1;
}
```

Exercise 6 There is a number game called Hotpo that can entertain you for a few minutes while you're stuck in traffic, waiting for a movie to start, or having dinner with someone really boring. Hotpo stands for Half Or Triple Plus One, and it works like this: Think of an odd number. Now mentally calculate another number, as follows: If the first number was even, the next number is half the first one; if the first number was odd, the next number is 3 times the first number, plus 1. Now you can forget the first number and apply the Half Or Triple Plus One formula to your current number. Keep going until the value reaches 1. Let's try this with a starting number of 5. The series is $5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$. Write a program that plays Hotpo. First, initialize a variable called `n` to the starting value you're interested in. Then enter a loop that prints out each number in the sequence, along with the current step number. For example, the output for 3 would be

```
Step #1: 10
Step #2: 5
Step #3: 16
Step #4: 8
Step #5: 4
Step #6: 2
Step #7: 1
```

Should the program use a while loop or a for loop?

Solution 6 Hotpo is an extreme example of a situation that ought to use a while loop. Remember that for loops are better when you know beforehand how many passes you will make through the loop's body, and while loops are better when you don't know you're done until you're done. If you've played with various values of n , you may have noticed that there's no way to predict whether a certain starting value will need a lot of steps or only a few steps to reach 1. Hotpo defies mathematical analysis. There seems to be no way to predict how many steps a given starting value will require, which means a while loop is ideal. Here is a solution:

```
public class HotpoWhile
{
    public static void main(String[] args)
    {
        int n = 3;
        int nSteps = 0;
        while (n != 1)
        {
            n = (n%2 == 0) ? n/2 : 3*n+1;
            nSteps++;
            System.out.println("Step #" + nSteps + ": " + n);
        }
    }
}
```

Note: If you're ever really bored, try 31.

Exercise 7 What is the value of n after the following code is executed?

```
int n = 1;
outer: for (int i=2; i<10; i++)
{
    for (int j=1; j<i; j++)
    {
        n *= j;
        if (i*j == 10)
            break outer;
    }
}
```

Solution 7 The answer is 24, but the real question is: How did you arrive at the answer? The code is only 10 lines long, and four of those lines are just curly brackets, but the nested loops are intricate enough that working out the answer mentally or on paper is unreliable. It doesn't take long to just type in the code, let it run, and see what happens.

Chapter 6

Exercise 1 The following two declarations are equivalent as far as the compiler is concerned, but one is considered more readable than the other. Which is more readable, and why?

- `double dubs[];`
- `double[] dubs;`

Solution 1 Format B (`double[] dubs`) is more readable than Format A (`double dubs[]`) because in B, as with all other declarations, the data type (`double[]`) comes first, followed by the variable name (`dubs`). Format A begins with some, but not all, of the data type (`double`). Then comes the variable name, followed by the remainder of the data type (`[]`). So Format A is less readable for two reasons: It does not follow the convention of data type followed by variable name, and it splits the data type into two parts.

Exercise 2 Write a line of code that declares an array of 5 ints and initializes the array to contain the first 5 prime numbers. The code should be a single statement.

Solution 2 `int[] first5Primes = {2, 3, 5, 7, 11};`

Exercise 3 Write a method whose single argument is an array of double. The method should return the average (mean) of the array's components. Write an application that tests the method by passing it an array containing any values you like.

Solution 3 The following code is one possible solution:

```
public class MeanOfArray
{
    public static void main(String[] args)
    {
        double[] theArray = {1.2, 1.3, 1.4, 1.5, 1.6};
        double average = computeAverage(theArray);
        System.out.println("mean = " + average);
    }

    static double computeAverage(double[] doubles)
    {
        double sum = 0;
        for (int i=0; i<doubles.length; i++)
            sum += doubles[i];
        return sum/doubles.length;
    }
}
```

Exercise 4 Write a program that uses the array-averaging method of Question 3. The program should compute and print out the average of an array (you can choose the component values). Then the program should add 100 to each component, and again compute and print out the average.

Solution 4 The following code is one possible solution. The `main` method is long enough that comments are in order:

```
public class Question4
{
    public static void main(String[] args)
    {
        // Create the array.
        double[] theArray = {1.2, 1.3, 1.4, 1.5, 1.6};

        // Compute and print out average.
        double average = computeAverage(theArray);
        System.out.println("mean = " + average);

        // Add 100 to each component.
        for (int i=0; i<theArray.length; i++)
            theArray[i] += 100;

        // Compute and print out new average.
        average = computeAverage(theArray);
        System.out.println("mean = " + average);
    }

    static double computeAverage(double[] doubles)
    {
        double sum = 0;
        for (int i=0; i<doubles.length; i++)
            sum += doubles[i];
        return sum/doubles.length;
    }
}
```

Exercise 5 Write a program that contains a method that creates and returns an array of int containing the first `n` square numbers, where `n` is the method's argument. Test your method by calling it with `n=10`. Your program should print out the index and value of each component, in *descending* order.

Solution 5 The following code is one possible answer. Note that the loop in `main` decrements its loop counter down to and including 0, because of the requirement that the squares should be printed out in descending order:

```
public class DescendingSquares
{
    public static void main(String[] args)
    {
        int[] squares = createArrayOfSquares(10);
        for (int i=squares.length-1; i>=0; i--)
            System.out.println(squares[i]);
    }

    static int[] createArrayOfSquares(int nSquares)
    {
        int[] squares = new int[nSquares];
        for (int i=0; i<nSquares; i++)
            squares[i] = i*i;
        return squares;
    }
}
```

Exercise 6 Write a method that creates a multiplication table. The method should return a two-dimensional array of N by N ints, where N is specified by the method's argument. In the array, the component at [row][col] should have a value of row*col.

Solution 6 The following method creates a multiplication table:

```
static int[][] makeTable(int n)
{
    int[][] table = new int[n][n];
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            table[i][j] = i*j;
    return table;
}
```

Chapter 7

Exercise 1 Name four traits that arrays and objects have in common.

Solution 1 Any four of the following are acceptable answers:

- They contain clusters of data.
- They are created by invoking the keyword `new`.
- They inhabit inaccessible memory.
- They are manipulated indirectly, via references.
- They cannot be passed as array method arguments, but references to them can.
- They are not destroyed explicitly. They are garbage-collected when they have no more references.

Exercise 2 Name two differences between arrays and objects.

Solution 2 Any two of the following are acceptable answers:

- They can contain data of different types.
- They can contain methods as well as data.
- They are related to classes.

Exercise 3 Objects are not passed as method arguments, but references to objects can be passed. When a reference is passed into a method, any changes made to the referenced object by the method should be visible to the method's caller. Write an application to demonstrate this.

Your application will have two classes: `Cat` and `Ager`. The `Cat` class should have a single variable: an `int` called `age`. The `Ager` class should have a method whose signature is `makeOlder(Cat kitty, int nYears)`. This method should add `nYears` to the age of the `Cat` object referenced by `kitty`. Your `main` method should go in the `Ager` class. It should create one instance of each class, set the cat's age, and then use the `Ager`'s method to change the age. Your `main` should then print out the cat's new age, and verify that it really changed.

Solution 3 In file `Cat.java`:

```
public class Cat
{
    int age;
}
```

In file `Ager.java`:

```
public class Ager
{
    void makeOlder(Cat kitty, int nYears)
    {
        kitty.age += nYears;
    }

    public static void main(String[] args)
    {
        Ager myAger = new Ager();
        Cat myCat = new Cat();
        myCat.age = 5;
        System.out.println("Age was " +
            myCat.age);
        myAger.makeOlder(myCat, 2);
        System.out.println("Age became " +
            myCat.age);
    }
}
```

Exercise 4 What happens if you move the `main` method of the previous question from the `Ager` class to the `Cat` class?

Solution 4 You have to run the program by typing "`java Cat`" instead of "`java Ager`". Otherwise the output is the same. The point of this question is that in a multiple-class application, you have to decide where to put your `main` method.

Exercise 5 Write an application that causes a "null pointer exception" failure.

Solution 5 The following application causes a "null pointer exception" failure:

```
public class IFail
{
    int x;

    public static void main(String[] args)
    {
        IFail ref = null;
        ref.x = 10;
    }
}
```

This is just one of an infinite number of possible examples. When you write long programs, "null pointer exception" failures are unavoidable in the course of developing, debugging, and refining your code.

Exercise 6 What does the following application print out?

```
public class Question
{
    static long x;

    public static void main(String[] args)
    {
        Question q1 = new Question();
        Question q2 = new Question();
        q1.x = 10;
        q2.x = q1.x + 20;
        System.out.println("q1.x = " + q1.x);
    }
}
```

Solution 6 The code prints out "q1.x = 30". Since x is static, "q1.x" and "q2.x" are both names for the same variable. The main method could be rewritten as

```
public static void main(String[] args)
{
    Question q1 = new Question();
    Question q2 = new Question();
    Question.x = 10;
    Question.x = Question.x + 20;
    System.out.println("q1.x = " + q1.x);
}
```

This question points out that referring to a static variable via the class name is clearer than referring to it via a reference to an instance of the class.

Chapter 8

Exercise 1 Which of the following hierarchies illustrate a good understanding of the difference between classes and objects? Which ones represent mistaken understanding? The arrows mean "has subclass", so in option A, Shape → Triangle means "class Shape has subclass Triangle."

- Shape → Triangle → RightTriangle
- GreatLiterature → GreatPoem → DivineComedy
- Planet → Continent
- Person → HeadOfState → Emperor
- Person → HeadOfState → Emperor → AugustusCaesar

Solution 1 A and D are good examples. "RightTriangle" is a category that falls within the broader category of "Triangle", which falls within the even broader category of "Shape". Similarly, "Emperor" is a category that falls within the broader category of "HeadOfState", which falls within the even broader category of "Person".

B starts off well: "GreatPoem" is a category that falls within the broader category of "GreatLiterature". But Dante's *Divine Comedy* is not a category. It is an instance of a category. In software, `divineComedy` should be an instance of class `GreatPoem`, which would be a subclass of `GreatLiterature`.

C isn't even close. Certainly, planets contain continents, and both planets and continents are categories of things, but a continent is not a more specific kind of planet. It would not be appropriate for class `Continent` to extend class `Planet`. (It might be appropriate for the two classes to exist, but be unrelated in terms of inheritance. In this case, perhaps `Continent` would have an array of `Planet`.)

E is like B. The last item is an instance of a category, not a category. Emperor is a category, but there was only one Augustus Caesar. So AugustusCaesar could be an instance of class `Emperor`, which extends class `HeadOfState`, which extends class `Person`.

Exercise 2 Which of the following classes have a no-args constructor?

- A)

```
class A { }
```
- B)

```
class B
{
    B() { }
```
- C)

```
class C
{
    C(int x) { }
```
- D)

```
class D
{
    D(int y) { }
    D() { }
```

Solution 2 There are two ways for a class to get a no-args constructor:

- It can define one explicitly.
- It can define no constructors at all. In that case, the compiler provides a default no-args constructor.

A has no constructors, so it is given a default no-args constructor. B and D define their own no-args constructors. C defines a constructor that takes arguments, so it has no no-args constructor.

Exercise 3 Write the code for two classes. The first, called `WaterBird`, has a float variable called `weight`. The class has a single constructor that looks like this:

```
WaterBird(float w)
{
    weight = w;
}
```

Compile this class. Now create the second class, called `Duck`, which extends `WaterBird`. `Duck` has no variables or methods, so it shouldn't take you long to write it. Will `Duck` compile? First, think about the issues involved. Then try to compile `Duck` and see if you were right.

Solution 3 The `Duck` class looks like this:

```
public class Duck extends WaterBird { }
```

It looks innocent enough, but if you've watched enough cartoons, you know that innocent-looking ducks are not to be trusted. This class defines no constructors, so it gets a default no-args constructor that does almost nothing. The constructor doesn't initialize anything (since it contains no code), but it does participate in the chain of construction. Thus, it tries to call the superclass's default constructor, and there we get into trouble.

The `WaterBird` superclass defines a constructor that takes an argument. There is no explicit no-args constructor, and there is no automatic default constructor. So an invisible piece of functionality in an invisible constructor in `Duck` is trying to call something in `WaterBird` that does not exist. When you try to compile `Duck`, you get an error message. The text of the message may vary depending on your compiler, but it will say something like this:

```
Constructor WaterBird() not found in class WaterBird
```

This kind of trouble is called the *constructor trap*. To get out of the trap, add a no-args constructor to the superclass.

Exercise 4 Write some code to demonstrate to yourself the chain of construction. Create an inheritance hierarchy of 4 classes. Give them any names you like. They don't have to have any data or methods, but each one should have a no-args constructor. These constructors should print out a line identifying the current class (something like "Constructing an instance of `WaterBird`"). Your `main()` method should construct a single instance of your lowest-level subclass. What is the output? Does it matter which class contains the `main()` method?

Solution 4 Here is one solution:

```
public class TwoDShape
{
    TwoDShape()
    {
        System.out.println("Constructor for TwoDShape");
    }
}

public class Polygon extends TwoDShape
{
    Polygon()
    {
        System.out.println("Constructor for Polygon");
    }
}

public class Triangle extends Polygon
{
    Triangle()
    {
        System.out.println("Constructor for Triangle");
    }
}

public class RightTriangle extends Triangle
{
    RightTriangle()
    {
        System.out.println("Constructor for RightTriangle");
    }

    public static void main(String[] args)
    {
        new RightTriangle();
    }
}
```

The output is

```
Constructor for TwoDShape
Constructor for Polygon
Constructor for Triangle
Constructor for RightTriangle
```

The application's behavior and output are the same no matter which class owns the `main()` method. However, it seems cleaner to put `main()` in `RightTriangle`. Anyone who reads the code for the first time will see the call to the `RightTriangle` constructor and wonder what class `RightTriangle` looks like. That person's job is easier if the `RightTriangle` class is the one they are already looking at.

In general, in a multiclass application you have some choices as to where to put your `main()` method. As always, think about which choice will be the clearest to someone reading the code for the first time.

Exercise 5 Write some code to demonstrate inheritance polymorphism. Create a superclass class with 3 subclasses. The superclass should have a method that prints out a line identifying the current class (something like "I am a Monster"). Two of the subclasses should override this method to print out a different message (like "I am a Werewolf"). Give the superclass a `main()` method with an array of size 4, typed as the superclass (for example, `Monster[] monsters = new Monster[4];`). Your `main()` should populate the array with references to 4 objects, each with a different class, and then traverse the array, calling your method on each array component. What is the output? Does it matter which class contains the `main()` method?

Solution 5 Here is one solution:

```
public class Monster
{
    void identify()
    {
        System.out.println("I am a monster.");
    }

    public static void main(String[] args)
    {
        Monster[] monsters = new Monster[4];
        monsters[0] = new Monster();
        monsters[1] = new Dragon();
        monsters[2] = new Werewolf();
        monsters[3] = new Cyclops();
        for (int i=0; i<monsters.length; i++)
            monsters[i].identify();
    }
}

public class Dragon extends Monster
{
    void identify()
    {
        System.out.println("I am a dragon.");
    }
}

public class Werewolf extends Monster
{
    void identify()
    {
        System.out.println("I am a werewolf.");
    }
}

public class Cyclops extends Monster
{
    void identify()
    {
        System.out.println("I am a cyclops.");
    }
}
```

The output is

```
I am a monster.
I am a dragon.
I am a werewolf.
I am a cyclops.
```

Again, programmatically it doesn't matter which class gets the `main()` method. As for readability, the major piece of data in `main()` is an array of `Monster`, so it makes sense to put `main()` in `Monster`.

Chapter 9

Exercise 1 Suppose package `superpack` contains subpackage `subpack`. Suppose a source file contains the following line:

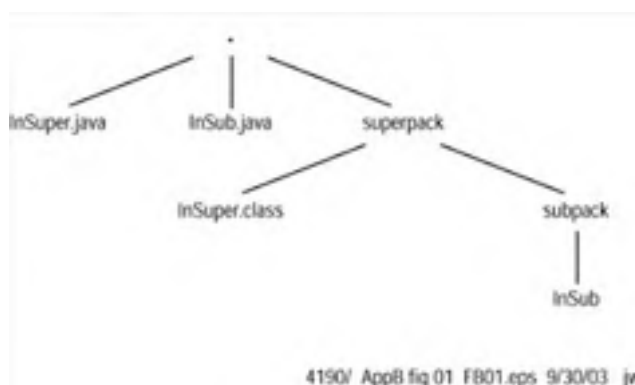
```
import superpack.*;
```

Will this line import classes in `subpack`? Write code to support your answer.

Solution 1 Let's start by creating two classes, one in each package:

```
package superpack;
public class InSuper { }
-----
package superpack.subpack;
public class InSub { }
```

These classes don't do anything, but they are all you need. You can store them in the same directory and compile by typing "`javac -d . *.java`", after which your directory looks like this:



Now you can see what the import line does. Create another class in yet another package:

```
package testpack;
import superpack.*;
public class TestClass
{
    InSub x;
}
```

The class declares a variable of type `InSub`. If the import line doesn't import contents of the subpackage, you should get a compiler error, because the compiler won't know what an `InSub` is. When you compile ("`javac -d . TestClass.java`"), you indeed get an error. This shows that importing "*" does not import subpackages.

Exercise 2 Create a class that illegally tries to read a private variable of another class. What is the point of this exercise?

Solution 2 First let's create and compile the class that owns the private variable:

```
class HasPrivate
{
    private int x;
}
```

Now to try to access `x`:

```
class AccessX
{
    void tryIt()
    {
        HasPrivate hp = new HasPrivate();
        hp.x = 10;
    }
}
```

When you try to compile this class, you get an error message that says something like

```
x has private access in HasPrivate
```

The point of this exercise, and the ones that follow, is to learn to recognize error messages that stem from misuse of the concepts in [Chapter 9](#). This will make it easier to fix bugs when they crop up later on. Meanwhile, you're also getting good practice at thinking in terms of packages and class inheritance structures.

Exercise 3 Create a class that illegally tries to call a default-access method of another class.

Solution 3 Your first class will be called `HasDefMethod`:


```
package aaaaa;
public class HasDefMethod
{
    void deffy()
    {
        System.out.println("deffy here.");
    }
}
```

The method may be called by any class in package aaaaa, so any illegal call attempt will have to come from a different package, like this:

```
package bbbbb;
import aaaaa.HasDefMethod;

public class BadCall
{
    void tryBadCall()
    {
        HasDefMethod h = new HasDefMethod(); // Ok
        h.deffy(); // Won't compile
    }
}
```

The compilation error message says something like this:

deffy() is not public in aaaaa.HasDefMethod; cannot be accessed from outside package

Exercise 4 Create a class that illegally tries to write a protected variable of another class.

Solution 4

You need to create a subclass in a different package from its superclass. First, here's the superclass:

```
package aaaaa;
public class HasProt
{
    protected double d;
} // And here's the subclass:
package bbbbb;
import aaaaa.HasProt;

public class BadWrite extends HasProt
{
    void misuse()
    {
        HasProt other = new HasProt();
        other.d = 3.14159; // Won't compile!
    }
}
```

The compilation error message is

d has protected access in aaaaa.HasProt

Since the subclass is in a different package from the superclass, an instance of the subclass may only access its own copy of a protected variable. In place of the line that doesn't compile, the following would be legal:

```
d = 3.14159;
```

Exercise 5 True or false: If a class has at least one abstract method, the class must be abstract. Write code to support your answer.

Solution 5 True. A class with any abstract methods must be abstract. The following class will not compile:

```
class NotAbstract
{
    abstract void abstractMethod();
}
```

Exercise 6 True or false: If a class is abstract, it must have at least one abstract method. Write code to support your answer.

Solution 6 False. It's okay for an abstract class to have no abstract methods. This isn't stated explicitly in the chapter, but it's easy enough to prove. The following class, which definitely doesn't contain any abstract methods, compiles without error:

```
Abstract class IsAbstract
{
}
}
```

Exercise 7 Write an application that tries to construct an instance of an abstract class. Can you compile the application? Can you execute it?

Solution 7 Here is an abstract class:

```
abstract class Ab { }
```

Here is an attempt to instantiate it:

```
class ConstructAbstract
{
    void constructInstanceOfAbstractClass()
    {

```

```
    Ab theInstance = new Ab();  
  }  
}
```

The compilation error message is something like this:

Ab is abstract; cannot be instantiated

Team LIB

◀ PREVIOUS

NEXT ▶

Chapter 10

Exercise 1 Suppose an interface declares three methods. And suppose a class declares that it implements the interface, but in fact it only implements two out of the three methods. What happens when you try to compile the class? (The way to answer this question, of course, is to write an interface and a class.)

Solution 1 You get a compilation error that says your class must be declared abstract. This is a perfectly sensible requirement. The following interface declares three methods:

```
interface Q1Inter
{
    public void a();
    public void b();
    public void c();
}
```

The following class does not completely implement the interface:

```
class Q1Class implements Q1Inter
{
    public void a()
    {
        System.out.println("Method a()");
    }

    public void b()
    {
        System.out.println("Method b()");
    }
}
```

When you compile, you get the following message or something very similar: "Class Q1Class should be declared abstract; it does not define method c() in interface Q1Inter."

Exercise 2 If class A implements an interface, any subclasses of A inherit all the methods specified in the interface. Does this mean that subclasses of A also implement the interface? Write code to discover the answer.

Solution 2 First, let's define the interface:

```
interface Q2Inter
{
    public void x();
}
```

Now here's a superclass that implements the interface:

```
class Q2Superclass implements Q2Inter
{
    public void x()
    {
        System.out.println("Hello from X.");
    }
}
```

And here's a subclass:

```
class Q2Subclass extends Q2Superclass
{
}
```

The subclass does not explicitly declare that it implements the interface, but it inherits an implementation of `x()` from its parent class. Does the compiler believe that `Q2Subclass` implements `Q2Inter`? Let's add some test code somewhere. We need a `main()` method, and we might as well put it in `Q2Subclass`:

```
class Q2Subclass extends Q2Superclass
{
    public static void main(String[] args)
    {
        Q2Subclass subby = new Q2Subclass();
        if (subby instanceof Q2Inter)
            System.out.println("It implements.");
        else
            System.out.println("It does not implement.");
    }
}
```

The application prints out "It implements", indicating that the subclass implicitly implements the interface declared explicitly by the superclass. In other words, interface implementation is a property that is inherited by subclasses.

Exercise 3 Given the following interface:

```
interface InterfaceQ3
{
    void printALine();
}
```

Will the following code compile?

```
class ClassQ3 implements InterfaceQ3
{
    void printALine()
    {
        System.out.println("OK");
    }
}
```

Solution 3 The code will not compile. The sources don't use explicit access modifiers. In the class code, this means `printALine()` has default access. But all methods (and constants) in an interface are public. The error message is something like this:

Method `printALint()` in class `ClassQ3` cannot implement method `printALint()` in interface `InterfaceQ3` w

Exercise 4 Don't worry, the following question requires absolutely no understanding of physics. In fact, it might make you grateful that you chose computer programming instead. Suppose you have the following interface:

```
package physics;
interface PhysicsConstants
{
    public static final double ELECTRON_MASS_KG = 9.11e-31;
    public static final double
        STEFAN_BOLTZMANN_CONSTANT_WATTS_PER_M2 = 5.67e-8;
}
```

What does the following application print out?

```
package physics;

public class Q4 implements PhysicsConstants
{
    public static void main(String[] args)
    {
        System.out.println("The value is " +
            STEFAN_BOLTZMAN_CONSTANT_WATTS_PER_M2);
    }
}
```

Solution 4 Trick question. The code doesn't print out anything, because it does not compile. There are two n's in "Boltzmann", but in the `main()` method there is only one.

The point of this question is to show that human eyes aren't the best mechanism for catching typos in long strings. When you try to compile the application, the compiler immediately finds the typo for you and directs you to the line you need to fix. If you used literal numerical values instead, you would be typing a much shorter string. `"5.67e-8"` only has 7 characters, versus 38 in `"STEFAN_BOLTZMANN_CONSTANT_WATTS_PER_M2"`, so the odds of a typo are 7/38 what they would be if you used the named constant. However, if you type `"5.67e-8"` enough times, you are bound to make a mistake eventually, and the effort of finding the typo would more than cancel out the time you saved by typing the shorter literal numeric value.

Chapter 11

Exercise 1 What happens when you run a program that creates an array of ints and then sets the value of an array component whose index is greater than the length of the array?

Solution 1 The following code tries to set component 60 in an array of length 50:

```
public class Ch11Q1
{
    public static void main(String[] args)
    {
        int[] ints = new int[50];
        ints[60] = 12345;
    }
}
```

The code compiles but throws an exception when executed. The exact message may vary, but the following is typical:

```
java.lang.ArrayIndexOutOfBoundsException at Ch11Q1.main(Ch11Q1.java:6) Exception in thread "main"
```

Exercise 2 What happens when you run a program that creates an array of ints whose length is less than zero?

Solution 2 The following code tries to set create an array of length -25:

```
public class Ch11Q2
{
    public static void main(String[] args)
    {
        int[] ints = new int[-25];
    }
}
```

The code compiles but throws an exception when executed. The exact message may vary, but the following is typical:

```
java.lang.NegativeArraySizeException at Ch11Q2.main(Ch11Q2.java:5) Exception in thread "main"
```

Exercise 3 What happens when you run a program that prints out the result of dividing a non-zero int by zero?

Solution 3 The following code tries to divide 39 by 0:

```
public class Ch11Q3
{
    public static void main(String[] args)
    {
        int and = 39 / 0;
    }
}
```

The code compiles but throws an exception when executed. The exact message may vary, but the following is typical:

```
java.lang.ArithmeticException: / by zero at Ch11Q3.main(Ch11Q3.java:5) Exception in thread "main"
```

Exercise 4 Write a program with a try block that just prints out a message. After the try block, add a catch block that catches `java.io.IOException` (which obviously is not thrown by the try block). Does the code compile? If it compiles, what happens when it runs?

Solution 4 The following code catches an exception type that is never thrown:

```
public class Ch11Q4
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Oye como va");
        }
        catch (java.io.IOException x)
        {
            System.out.println("Caught it.");
        }
    }
}
```

The Java compiler protects you from writing code that can never be run. Since `IOException` is not thrown from the try block, the catch block will never execute. Compilation fails with the following sort of error message:

```
Exception IOException is never thrown in the corresponding try block
```

Exercise 5 Suppose a try block throws many different subclasses of `IOException` (and no other exception types). Suppose you want to catch a few specific subclass types, such as `PrinterIOException` or `ConnectException`. All other exception types should be caught in a safety-net block. Your safety-net block can catch `IOException` or `Exception`. The code will produce the same behavior either way, but the "Catch Blocks and instanceof" section of [Chapter 11](#) says that it's better to use `IOException`. Speculate on why this is true.

Solution 5 Consider a stranger reading your program for the first time. Ask yourself how you can make the source code as easy as possible to understand. This is always a good thing to do, because one day that stranger might be you. (Even if you're reading your own code. It's amazing how you can come back to something you wrote only a few months ago, only to find that you don't remember why you did what you did.)

If your safety-net code catches `IOException`, the stranger will conclude, "The try block throws many kinds of `IOException`." But if your safety-net code catches `Exception`, the stranger will think, "The try block might throw *anything*." So a more specific safety-net catch block gives the stranger more specific information about what the try block might throw.

Exercise 6 What three decisions do you have to make when creating a custom exception subclass?

Solution 6 You have to decide if you want a checked exception or a runtime exception. You have to choose a name for your class. And you have to choose a superclass.

Team LiB

◀ PREVIOUS

NEXT ▶

Chapter 12

Exercise 1 In the beginning of [Chapter 12](#), you learned that a good rule of thumb is to use core code when you can and develop original code when you must. Because Java is an object-oriented language, you have a third option, which combines reusing existing code with creating your own. You learned about this option in an earlier chapter. What is it?

Solution 1 The third option is to subclass an existing class. The subclass you create combines preexisting features inherited from the superclass with new features that you implement in the subclass.

Exercise 2 If you write code that calls a deprecated method of one of the core Java classes, what valuable feature of Java can you no longer rely on?

Solution 2 Backward compatibility.

Exercise 3 Suppose you are reading someone else's code and you come across the following lines:

```
Stack myStack = new Stack(); // java.util package
myStack.setSize(100);
```

You decide to look up `setSize()` in the APIs. The comment kindly tells you that class `Stack` is in package `java.util`, so you click on `java.util` in the packages frame, and then you click on `Stack` in the classes frame. You find yourself looking at the class description. You scroll down to the method summaries, and you don't see `setSize` anywhere.

How should you proceed?

Solution 3 There are two ways that class `Stack` can get a `setSize()` method: it can implement it, or it can inherit it. Clearly `Stack` doesn't implement `setSize()`, so it must inherit it.

Scroll down past the end of the method summary section, to the inherited method section. You will see a list of methods inherited from `java.util.Vector`, which is `Stack`'s immediate superclass. There you will see a "setSize" link. Click on it to see the method description on the `java.util.Vector` page.

Alternately, you can scroll up to the top of the `Stack` description page to the inheritance hierarchy. There you will find a link to the `Vector` superclass. Scroll down to the method summaries, where you will find `setSize()`.

Exercise 4 In the section on the `String` class, you learned about the `startsWith(String s)` method, which returns `true` if the executing string object begins with the argument string `s`. It stands to reason that there should be a similar method that tells you whether the executing string object ends with a specified string. Look at the API page for `java.lang.String` and see if such a method exists.

Solution 4 The method does exist. It is called `endsWith()`.

Exercise 5 What happens when you try to compile and execute the following application?

```
public class Ch12Q5
{
    public String toString()
    {
        return "I am an instance of Ch12Q5.";
    }
    public static void main(String[] args)
    {
        Ch12Q5 thing = new Ch12Q5();
        System.out.println(thing);
    }
}
```

Solution 5 The program compiles and executes without error. The call to `System.out.println()` calls `toString()` on `thing`, so the output is

```
I am an instance of Ch12Q6.
```

Exercise 6 What happens when you try to compile and execute the following application?

```
class Ch12Q6
{
    String toString()
    {
        return "I am an instance of Ch12Q6.";
    }
    public static void main(String[] args)
    {
        Ch12Q6 thing = new Ch12Q6();
        System.out.println(thing);
    }
}
```

Solution 6 The difference between this application and the one in Exercise 5 is that the "public" modifiers have been removed from the declarations of the class and the `toString()` method. Now the code won't compile, because `toString()` is public in class `Object`, which is the superclass of `Ch12Q6`. If you override a method, as you have done here with `toString()`, it is illegal to give the subclass version weaker access than the superclass version.

Exercise 7 Look up the explanation of the `equals()` method on the API page of class `java.lang.Object`. The explanation is a bit wordy, but see if you can figure out what it does. (Focus on the last sentence, just before the "Parameters" section.) What is the technical term for what the method does? (Hint: It was introduced in [Chapter 12](#).)

Solution 7 The method checks for reference equality. This isn't so useful, because `equals()` is supposed to check for object equality. No wonder subclasses of `Object` override `equals()`.

Exercise 8 You're not allowed to construct an instance of the `java.lang.Math` class. What happens if you try?

Solution 8 If you write a program that contains the line

```
Math m = new Math();
```

you will get an error message that says something like

```
...constructor Math() has private access in class java.lang.Math.
```

The constructor for the `java.lang.Math` is private. This means that the constructor can be invoked only from within the class itself. This is how the class ensures that you and I can never write code that constructs a `Math` instance.

Exercise 9 The following code models the behavior of a familiar piece of equipment that is used in many games throughout the world. What is the piece of equipment?

```
long rand = 1 + Math.round(Math.random() * 5);
```

Solution 9 The code generates a random int that is ≥ 1 and ≤ 6 , so it simulates shaking dice.

Chapter 13

Exercise 1 In [Chapter 13](#), you learned about the following line:

```
String s = "C:my_backup\temporary\news";
```

What does the following code print out?

```
String s = "C:my_backup\temporary\news";  
System.out.println("***\n" + s + "***");
```

What is the moral of this exercise?

Solution 1 The code prints the following bizarre output:

```
***
```

```
C:my_backup      emporary  
ews
```

The backslash-t is interpreted as a tab, and the backslash-n is interpreted as a newline. The moral is that you always have to use double backslashes in literal strings and chars if you want an actual backslash and not an escape code.

Exercise 2 The code examples in the "[Writing and Reading Data](#)" section defined an int called `i`, a float called `f`, a double called `d`, and so on. But the long was called `n`, which breaks the pattern. You might have expected the long to be called `l`. Why do you think this was not done?

Solution 2 The lowercase letter "ell" looks just like a "one." If you use a lowercase "ell" as a variable name, your code becomes hard to understand. Uppercase "oh" and lowercase "ell" are the two least readable variable names. (Notice how they are spelled out here, in order to make sure there is no confusion. It would be less helpful to tell you that O and I are bad variable names.)

Exercise 3 Write a program that creates a file containing 5,000 random doubles that are ≥ 0 and < 200 .

Solution 3 The following code creates a file that contains 5,000 random numbers in the required range:

```
import java.io.*;  
  
public class Ch13Q3  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            FileOutputStream fos;  
            DataOutputStream dos;  
            fos = new FileOutputStream("RandomDoubles");  
            dos = new DataOutputStream(fos);  
            for (int i=0; i<5000; i++)  
            {  
                double randy = Math.random() * 200;  
                dos.writeDouble(randy);  
            }  
            dos.close();  
            fos.close();  
        }  
        catch (IOException x)  
        {  
            System.out.println("Caught IOException");  
        }  
    }  
}
```

Exercise 4 Write a program that verifies the file you created in the previous exercise. Your program should read the 5,000 doubles, making sure that each falls within the proper range. Your program should also make sure the file contains exactly 5,000 longs.

Solution 4 The following code validates the file that was created in Exercise 3:

```
import java.io.*;  
  
public class Ch13Q4  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            FileInputStream fis =  
                new FileInputStream("RandomDoubles");  
            DataInputStream dis = new DataInputStream(fis);  
            boolean readBad = false;  
            for (int i=0; i<5000; i++)  
            {  
                double randy = dis.readDouble();  
                if (randy < 0 || randy > 200)  
                {  
                    readBad = true;  
                    System.out.println("Read bad double: " +  
                        randy);  
                }  
            }  
        }  
        catch (IOException x)  
        {  
            System.out.println("Caught IOException");  
        }  
    }  
}
```

```
                randy);
            }
        }
        if (!readBad)
            System.out.println("File is valid.");
    }

    catch (IOException x)
    {
        System.out.println("Caught IOException");
    }
}
}
```

The for loop reads 5,000 doubles from the file and checks their range. If a double is out of range, the "Read bad double" message is printed out and the variable `readBad` is set to `true`. After the loop, `readBad` is checked. If it never got set to `true`, the file is considered valid.

Exercise 5 Look up the API documentation for the `java.io.File` class. An instance of this class contains information about an individual file. One of the methods of the class tells you the length in bytes of a file. Use this method to determine the number of bytes in the file you created in Exercise 3.

Solution 5 The following program uses the `File` class to read the size of the file created in Exercise 3:

```
import java.io.*;

public class Ch13Q5
{
    public static void main(String[] args)
    {
        File f = new File("RandomDoubles");
        System.out.println("Length = " + f.length());
    }
}
```

The code prints out "Length = 40000". This is to be expected. The file contains 500 doubles, and each double is 8 bytes.

Chapter 14

Exercise 1 The first code example in [Chapter 14](#) used the following code to set a frame's background color:

```
setBackground(new Color(128, 128, 128));
```

Describe the color that this line creates.

Solution 1 The color has equal levels of red, green, and blue, so it will be some kind of gray. Since the levels are halfway between the minimum (0) and the maximum (255), the gray will be about halfway between black and white: a neutral gray, neither dark nor light.

Exercise 2 Run Color Lab, and adjust the scrollbars so that the displayed color matches something you can see (a piece of clothing you're wearing, or something on your desk, or anything else you like). Now write an application that displays a frame whose interior is the color you've chosen.

Solution 2 The following code shows a frame whose interior matches the color of the shirt I was wearing when I wrote this.

```
import java.awt.*;

public class EmptyFrame extends Frame
{
    EmptyFrame()
    {
        setBackground(new Color(0, 217, 255));
        setSize(300, 300);
    }

    public static void main(String[] args)
    {
        EmptyFrame em = new EmptyFrame();
        em.setVisible(true);
    }
}
```

Remember that in addition to setting the background color, you have to call `setSize()` and `setVisible()`. Otherwise the frame cannot be seen.

Exercise 3 One of the code examples in [Chapter 14](#) used the `getSize()` method, which `Frame` inherits from one of its superclasses. Use the API to find out which superclass implements the method.

Solution 3

```
java.awt.Component
```

Exercise 4 Write a program that draws a five-pointed star. Your frame should be 400 x 400 pixels. The coordinates of the star's points are (200, 375), (97, 58), (366, 254), (34, 254), and (303, 58). The easy way is to write a `paint()` method that calls `drawLine()` five times. But that approach isn't ideal, because you have to type each x and each y twice. (Each point is the end of two lines, so it appears in two `drawLine()` calls.) Typing data, code, or anything else more than once is considered bad style. If one of the copies has a typo and doesn't match the original precisely, your program won't function correctly. To avoid duplication of data, your program should have two `int` arrays, defined as follows:

```
int[] xs = {200, 97, 366, 34, 303};
int[] ys = {375, 58, 254, 254, 58};
```

Your `paint()` method should have a loop that accesses these arrays. `drawLine(...)` should appear only once in your code, inside the loop.

Solution 4 The following program uses a loop to draw a five-pointed star:

```
1. import java.awt.*;
2.
3. public class Supe extends Frame
4. {
5.     int[] xs = {352, 106, 200, 294, 48};
6.     int[] ys = {151, 329, 40, 329, 151};
7.
8.     Supe()
9.     {
10.        setSize(400, 400);
11.    }
12.
13.    public void paint(Graphics g)
14.    {
15.        for (int startPoint=0;
16.            startPoint<xs.length;
17.            startPoint++)
18.        {
19.            int endPoint = startPoint + 1;
20.            if (endPoint == xs.length)
21.                endPoint = 0;
22.            g.drawLine(xs[startPoint], ys[startPoint],
23.                    xs[endPoint], ys[endPoint]);
24.        }
}
```

```
25.     }
26.
27.     public static void main(String[] args)
28.     {
29.         (new Supe()).setVisible(true);
30.     }
31. }
```

Lines 19-21 can be replaced by the following single line:

```
Int endPoint = (startPoint+1) % xs.length;
```

Exercise 5 Write a program that lists all the font families that are available on your computer.

Solution 5 The following program lists all the available font families.

```
import java.awt.GraphicsEnvironment;

public class ListFonts
{
    public static void main(String[] args)
    {
        GraphicsEnvironment grenv =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        String[] names =
            grenv.getAvailableFontFamilyNames();
        for (int i=0; i<names.length; i++)
            System.out.println(names[i]);
    }
}
```

Chapter 15

Exercise 1 Suppose you use the following code to create a checkbox:

```
Checkbox cbox = new Checkbox("Ok", true);
```

What is the checkbox's state after you click on it 20,000 times?

Solution 1 The checkbox's initial state is `true`. After you click on it an even number of times, it is `true` again. There are two ways to get the answer: thinking about it, or doing it. If you chose the second way, you might not be cut out to be a computer programmer.

Exercise 2 In the "Checkboxes" section of [Chapter 15](#), the `Boats` application is 30 lines long. The code isolates literal strings in an array near the top of the listing. You saw how this approach, along with the use of a loop to create the checkboxes, results in more maintainable code. Rewrite the code to eliminate the loop and the string array. In place of the loop in the constructor, just create three checkboxes one by one. How many lines of code does your new application have?

Solution 2 Here is the rewritten code:

```
1. import java.awt.*;
2.
3. class BoatsNoLoop extends Frame
4. {
5.     Checkbox[] cboxes;
6.     Button btn;
7.
8.     BoatsNoLoop()
9.     {
10.         setLayout(new FlowLayout());
11.
12.         cboxes = new Checkbox[3];
13.         cboxes[0] = new Checkbox("a small boat");
14.         add(cboxes[0]);
15.         cboxes[1] = new Checkbox("a medium boat");
16.         add(cboxes[1]);
17.         cboxes[2] = new Checkbox("a large boat");
18.         add(cboxes[2]);
19.         btn = new Button("Add to shopping cart");
20.         add(btn);
21.
22.         setSize(600, 200);
23.     }
24.
25.     public static void main(String[] args)
26.     {
27.         new BoatsNoLoop().setVisible(true);
28.     }
29. }
```

This version is only 29 lines, one line shorter than the original. The point is that a shorter program is not necessarily easier to read or maintain than a longer version. If you still aren't convinced that the original version is better, try Exercise 3.

Exercise 3 This is an extension of Exercise 2. Suppose you need to change the `Boats` application so that instead of offering three sizes (small, medium, and large), it offers ten (rubber duck, sponge, tiny, small, kinda small, medium, kinda large, large, huge, titanic). How does this affect the size of the code as it appears in the "Checkboxes" section of [Chapter 15](#)? How does it affect the size of the code that you wrote for Exercise 2?

Solution 3 The loop-based code will probably become longer by two lines, because the array of literal strings now has 10 members:

```
String[] sizes = {"rubber duck", "sponge", "tiny", "small", "kinda small",
"medium", "kinda large", "large", "huge", "titanic"};
```

The no-loop version grows by two lines for every additional size option (one line to construct a checkbox, another to call `add()`). Seven new options were added, so the code grows by 14 lines, or 50%.

Note Programs usually start out small, and gradually grow as they are required to support more and more functionality. Giving structure to a small program is always worth the effort. The payoff might not be immediately obvious, but it will become more and more evident as time goes by. In Exercise 2, the well-structured program was actually longer than the unstructured version. But when the code needed to support more functionality, the unstructured version grew by 50% while the structured version grew by about 7%.

Exercise 4 Write an application that displays a frame with a menu bar. The bar should have the following menus:

An Edit menu with items Copy and Cut.

A File menu with items Close, Exit, and Open.

A Help menu with item Help. Assume that clicking on this item will display a helpful dialog.

A Whatever menu with items Stuff and Nonsense. The Nonsense item should be a submenu with items Ordinary Nonsense and Extreme Nonsense.

Make sure that your GUI follows the guidelines listed at the end of the "Menus" section.

Solution 4 Here is one solution:

```
import java.awt.*;

class Q4 extends Frame
{
    public Q4()
    {
        MenuBar mbar = new MenuBar();

        Menu fileMenu = new Menu("File");
        fileMenu.add("Open...");
        fileMenu.add("Close");
        fileMenu.add("Exit");
        mbar.add(fileMenu);

        Menu editMenu = new Menu("Edit");
        editMenu.add("Cut");
        editMenu.add("Copy");
        mbar.add(editMenu);

        Menu whateverMenu = new Menu("Whatever");
        whateverMenu.add("Stuff");
        Menu nonsenseMenu = new Menu("Nonsense");
        nonsenseMenu.add("Ordinary Nonsense");
        nonsenseMenu.add("Extreme Nonsense");
        whateverMenu.add(nonsenseMenu);
        mbar.add(whateverMenu);

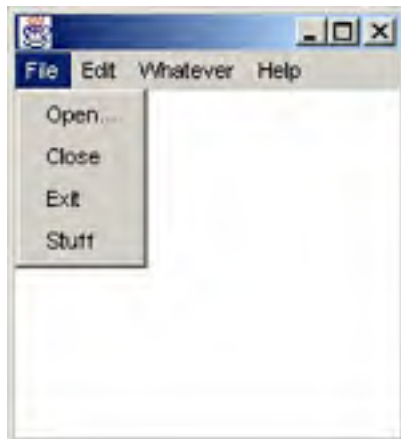
        Menu helpMenu = new Menu("Help");
        helpMenu.add("Help...");
        mbar.add(helpMenu);

        setMenuBar(mbar);

        setSize(300, 200);
    }

    public static void main(String[] args)
    {
        (new Q4()).setVisible(true);
    }
}
```

The following illustrations show the four menus, File, Edit, Whatever, and Help:





Exercise 5 Write a program that creates a GUI that looks like the following illustration. The text in the text area should be set programmatically by a single call to the text area's `append()` method. The call should come directly after the text area is constructed.



Solution 5 Here is one solution:

```
1. import java.awt.*;
2.
3. class Q5 extends Frame
4. {
5.     public Q5()
6.     {
7.         setLayout(new FlowLayout());
8.         TextArea ta = new TextArea(10, 30);
9.         ta.append("Hello\nWorld");
10.        add(ta);
11.
12.        setSize(550, 220);
13.    }
14.
15.    public static void main(String[] args)
16.    {
17.        (new Q5()).setVisible(true);
18.    }
19. }
```

The text is set in line 9. The thing to notice is the newline character, which puts in a line break. It would be wrong to replace line 9 with this:

```
ta.append("Hello");
ta.append("World");
```

Line breaks are inserted only when you explicitly append a newline character. So with the substitution, you would see a single line of text that read "HelloWorld". There wouldn't even be a space between the words.

Exercise 6 Using the API page for `java.awt.FlowLayout`, determine how to create a flow layout manager that right-justifies its cluster of components rather than centering it.

Solution 6 Use the following constructor:

```
new FlowLayout(FlowLayout.RIGHT)
```

Exercise 7 The `java.awt.Component` class, which is a superclass of `java.awt.Button`, has a method called `setSize(int width, int height)`. The method's documentation says that it resizes the component so that its size is `width` times `height`.

What do you expect the following code to do? First, read the listing and decide on your answer. Then, type in the code and run it. Did you see what you expected to see?

```
import java.awt.*;

class Q7 extends Frame
{
    public Q7()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Abcde");
        btn.setSize(500, 500);
        add(btn);
        setSize(700, 700);
    }

    public static void main(String[] args)
    {
        (new Q7()).setVisible(true);
    }
}
```

Solution 7 The code seems to create a large (500 x 500) button in a 700 x 700 frame. But actually, the button's size is perfectly ordinary. The Flow layout manager sets the size of the button to its preferred size.

Exercise 8 This entire chapter has been about components that are installed inside containers. The [previous chapter](#) was about painting. What happens if a frame that contains components also has a `paint()` method that paints a part of the screen that is occupied by a component? Write a program that will reveal the answer.

Solution 8 The frame in the following application has a button, as well as a `paint()` method. The `paint()` method draws diagonal blue lines.

```
import java.awt.*;

class PaintPlusComponent extends Frame
{
    public PaintPlusComponent()
    {
        setLayout(new FlowLayout());
        add(new Button("Apply"));
        setSize(300, 200);
    }

    public void paint(Graphics g)
    {
        g.setColor(Color.blue);
        for (int i=0; i<500; i+=10)
            g.drawLine(i, 0, 0, i);
    }

    public static void main(String[] args)
    {
        (new PaintPlusComponent()).setVisible(true);
    }
}
```

The following illustration shows the GUI. As you can see, the button is superimposed over the painted lines. Whenever a component and a `paint()` method are both responsible for the same part of the screen, the component wins.



Chapter 16

Exercise 1 Write a program that displays a frame. The frame's `paint()` method should draw something simple. The application should also maintain a count of the number of times `paint()` is called. This count should be printed out every time `paint()` is called. Execute your application, and use it to help determine whether `paint()` is called when:

The application starts up.

The frame is minimized/iconified.

The frame is restored to normal size after being minimized/iconified.

The frame is restored to normal size after being minimized/iconified.

The frame is moved.

The frame is partially covered by another frame.

The frame is uncovered.

Solution 1 The following code prints a message whenever `paint()` is called:

```
import java.awt.*;

public class Ch16Q1 extends Frame
{
    int    nCallsToPaint;

    Ch16Q1()
    {
        setSize(300, 300);
    }

    public void paint(Graphics g)
    {
        g.setColor(Color.cyan);
        g.drawLine(100, 100, 200, 200);
        nCallsToPaint++;
        System.out.println(nCallsToPaint +
                           " calls to paint()");
    }

    public static void main(String[] args)
    {
        (new Ch16Q1()).setVisible(true);
    }
}
```

The `paint()` method is called when the application starts up, and when it is restored after being minimized/iconified. It is also called when the frame is uncovered. It is *not* called when the frame is moved. Depending on your system, it may or may not be called when the frame is covered.

Exercise 2 Every Java thread is represented by an instance of the `java.lang.Thread` class. You can get a reference to the currently running thread by calling the `currentThread()` static method of the `Thread` class. Threads have names. The class has a method called `getName()`, which returns the name as a string. So you can print out the name of the current thread by calling

```
System.out.println(Thread.currentThread().getName());
```

Write a simple frame application that makes this call in its `main()` method and in its `paint()` method. Verify that `main()` and `paint()` are executed in different threads.

Solution 2 The following application prints the name of the current thread in `main()` and `paint()`:

```
import java.awt.*;

public class Ch16Q2 extends Frame
{
    Ch16Q2()
    {
        setSize(300, 300);
    }

    public void paint(Graphics g)
```

```
{
    g.setColor(Color.cyan);
    g.drawLine(100, 100, 200, 200);
    System.out.println("paint() thread is called:");
    System.out.println(Thread.currentThread().getName());
}

public static void main(String[] args)
{
    System.out.println("main() thread is called:");
    System.out.println(Thread.currentThread().getName());
    (new Ch16Q2()).setVisible(true);
}
}
```

Exercise 3 Write an application that adds the same action listener to a button *twice*. For example, if `myButton` is the button and `myListener` is the action listener, your code would contain the following lines:

```
myButton.addActionListener(myListener);
myButton.addActionListener(myListener);
```

Your listener's `actionPerformed()` method should print out a message to tell you that it got called. If you press the button once, do you expect the message to be printed out once or twice? Run your application to see if you guessed right.

Of course, in real life there would never be a good reason for doing this. But you might do it by accident. For example, you might paste the line into your source code twice by accident. So it's good to know in advance what the symptom will be, so that you can recognize it and fix the problem if it ever comes up.

Solution 3 Here is the listener class:

```
import java.awt.event.*;

class Aclis implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("actionPerformed() was called.");
    }
}
```

And here is the application class:

```
import java.awt.*;

public class Ch16Q3 extends Frame
{
    Ch16Q3()
    {
        setLayout(new FlowLayout());
        Button btn = new Button("Push Me");
        Aclis ac = new Aclis();
        btn.addActionListener(ac);
        btn.addActionListener(ac);
        add(btn);
        setSize(300, 300);
    }

    public static void main(String[] args)
    {
        (new Ch16Q3()).setVisible(true);
    }
}
```

When you push the button, the message is printed out twice.

Exercise 4 Suppose a class has an `actionPerformed()` method, as specified by the `ActionListener` interface, but the class does not state that it implements the interface. Can an instance of the class be used as a button's action listener?

Solution 4 The following class contains an `actionPerformed()` method, but it does not declare that it implements the `ActionListener` interface:

```
import java.awt.event.*;

class NotAnActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("actionPerformed() was called.");
    }
}
```

Since the class has the right kind of method, you might be tempted to use it as an action listener:

```
. . .
Button btn = new Button("OK");
NotAnActionListener naal = new NotAnActionListener();
btn.addActionListener(naal);
. . .
```

This code will not compile. For a class to be eligible to be an action listener, it is not enough for it to provide an `actionPerformed()` method, because that alone does not mean that it implements the `ActionListener` interface.

Exercise 5 Run Nim Lab by typing `java events.NimLab`. Select Disable Buttons..... and play the game. This version is the result of three rounds of improvements made to the original program. What additional improvements can you suggest? Think about how the game could be modified to make the GUI easier and more natural.

Solution 5 Here are some possible improvements:

- Add a Restart button.
- Provide notification when a player wins.
- Eliminate the buttons. Players would click on a coin to remove it. This would provide direct manipulation of the coins, rather than the indirect manipulation that the buttons provide.

Do you have any other ideas? E-mail them to groundupjava@squares.com, and they might be included in the next revision of this book (with your name mentioned).

Exercise 6 The various event classes (`ActionEvent`, `ItemEvent`, etc.) all inherit the `getSource()` method from a superclass. Use the API pages to determine the name of that superclass.

Solution 6 `java.util.EventObject`.

Exercise 7 Write an application with a GUI that contains a choice and a text area. When the choice is activated, a message should be written to the text area, stating the choice's selected index.

Suggested design: Your frame should contain a panel (at North) that contains the choice. The text area should be at South. If you need a guideline, the `TextAreaNim` program in the "Improving the GUI" section has a similar structure.

Solution 7 Here's the code:

```
import java.awt.*;
import java.awt.event.*;

public class Ch16Q7 extends Frame implements ItemListener
{
    private Choice    choice;
    private TextArea ta;

    Ch16Q7()
    {
        Panel pan = new Panel();
        choice = new Choice();
        choice.add("Dragons");
        choice.add("Centaur");
        choice.add("Unicorns");
        choice.add("Manticores");
        choice.addItemListener(this);
        pan.add(choice);
        add(pan, "North");

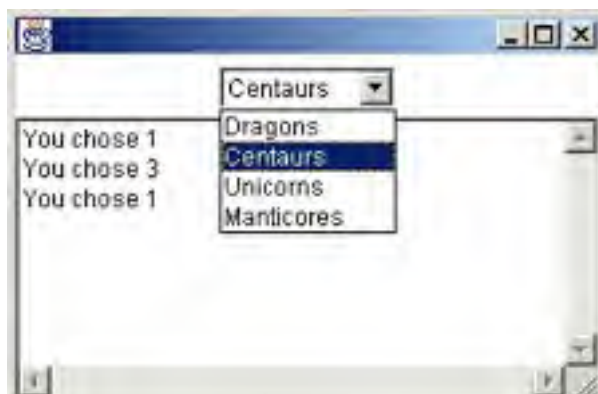
        ta = new TextArea(40, 20);
        add(ta, "Center");

        setSize(300, 200);
    }

    public void itemStateChanged(ItemEvent e)
    {
        ta.append("You chose " +
                choice.getSelectedIndex() +
                "\n");
    }

    public static void main(String[] args)
    {
        (new Ch16Q7()).setVisible(true);
    }
}
```

The following illustration shows the GUI for Exercise 7.



Exercise 8 Write an application with a GUI that contains a text field and a text area. When the user presses the Enter key in the text field, the text field's contents should be copied into text area, followed by a newline character.

Your event-handling code will need to retrieve the contents of the text field. You do that by calling the text field's `getText()` method, which returns a string.

Suggested design: Your frame should contain a panel at North that contains the text field. The text area should go at Center.

Solution 8 Here's the code:

```
import java.awt.*;
import java.awt.event.*;

public class Ch16Q8 extends Frame implements ActionListener
{
    private TextField    tf;
    private TextArea    ta;

    Ch16Q8()
    {
        Panel pan = new Panel();
        tf = new TextField("Type Here", 20);
        tf.addActionListener(this);
        pan.add(tf);
        add(pan, "North");

        ta = new TextArea(40, 20);
        add(ta, "Center");

        setSize(300, 200);
    }

    public void actionPerformed(ActionEvent e)
    {
        ta.append(tf.getText() + "\n");
    }

    public static void main(String[] args)
    {
        (new Ch16Q8()).setVisible(true);
    }
}
```

The following illustration shows the GUI for Exercise 8.



Chapter 17

Exercise 1 Write a program that creates a frame with a File menu. The menu should have two items, Save... and Exit. When Save... is selected, the code should display a file dialog box, configured for saving a file. When the user has specified a file via the dialog box, your code should output the name of the file. All the information you need is on the API page for `java.awt.FileDialog`.

Solution 1 Here's the code:

```
import java.awt.*;
import java.awt.event.*;

class SaverFrame extends Frame implements ActionListener
{
    private MenuItem saveMI, exitMI;

    public SaverFrame()
    {
        // Build menu.
        MenuBar mbar = new MenuBar();
        Menu fileMenu = new Menu("File");
        saveMI = new MenuItem("Save...");
        saveMI.addActionListener(this);
        fileMenu.add(saveMI);
        exitMI = new MenuItem("Exit");
        exitMI.addActionListener(this);
        fileMenu.add(exitMI);
        mbar.add(fileMenu);
        setMenuBar(mbar);
        setSize(300, 150);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == exitMI)
            System.exit(0);

        FileDialog dia = new FileDialog(this, "Save Your Work",
                                       FileDialog.SAVE);

        dia.setVisible(true);
        String fileName = dia.getFile();
        if (fileName == null)
            System.out.println("You canceled the dialog.");
        else
            System.out.println("You chose file " + fileName + "
                               in " + dia.getDirectory());
    }

    public static void main(String[] args)
    {
        (new SaverFrame()).setVisible(true);
    }
}
```

The following illustration shows the file dialog, configured for saving.



Exercise 2 The `FileDialog` class has a `setDirectory()` method that controls which directory the dialog box will display. Look up the method description in the API to become familiar with how it works. Modify the final project code so that when the file dialog box appears, it displays one of the directories on your computer where you have stored some of your own Java source code. This will make it easier to display your own work.

Solution 2 Let's say you want the dialog to display the directory `C:\MyCode\Ch7_Exercises`. In `actionPerformed()`, change the code that constructs the file dialog, which in its original form looks like this:

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == openMI)
    {
        if (dialog == null)
            dialog = new FileDialog(this, "Source File",
                                   FileDialog.LOAD);

        dialog.setVisible(true); // Modal
    }
    ...
}
```

Add the `setDirectory()` call immediately after the dialog is constructed, before it is made visible:

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == openMI)
    {
        if (dialog == null)
        {
            dialog = new FileDialog(this, "Source File",
                                   FileDialog.LOAD);
            dialog.setDirectory("C:\\MyCode\\Ch7_Exercises");

            dialog.setVisible(true); // Modal
        }
    }
    ...
}
```

Tip Remember that in Java literal strings, single backslashes are escape characters that have special significance. That's why the argument to the `setDirectory()` call is `C:\\MyCode\\Ch7_Exercises` and not `C:\MyCode\Ch7_Exercises`.

Exercise 3 Write an application that displays a canvas subclass in a frame, at Center. The frame does not contain any other components.

Use the following code as the `paint()` method for the canvas subclass:

```
1. public void paint(Graphics G)
2. {
3.     g.setFont(new Font("Serif", Font.PLAIN, 24));
4.     g.setColor(Color.blue);
5.     g.drawString("Look at this!", 0, 0);
6. }
```

Run the program. Do you see what you expected to see? How do you explain the results?

Now change line 5 to this:

```
g.drawString("A bluejay in a quagmire", 0, 0);
```

Now do you see what you expected to see? Again, how do you explain the results?

Solution 3 Here's the code:

```
import java.awt.*;

class TextCanvas extends Canvas
{
    public void paint(Graphics g)
    {
        g.setFont(new Font("Serif", Font.PLAIN, 24));
        g.setColor(Color.blue);
        g.drawString("A bluejay in a quagmire", 0, 0);
    }

    public static void main(String[] args)
    {
        Frame fr = new Frame();
        TextCanvas tc = new TextCanvas();
        fr.add(tc, "Center");
        fr.setSize(250, 250);
        fr.setVisible(true);
    }
}
```

The y-coordinate argument of the `drawString()` method of class `Graphics` specifies the vertical position of the *baseline* of the text. If the baseline is 0, you will only see those parts of the text that descend below the baseline. In the string "Look at this!", there are no descenders. In "A jay in a quagmire", there is one occurrence of each character that descends: j, y, q, and g. The following illustration shows the GUI with the misplaced baseline. You can see the bottom portions of those letters, hanging down from the top of the canvas.



Exercise 4 The `FancySrcCanvas` class has an array of Java keywords. In that array, `throws` comes before `throw`. Otherwise, the list is alphabetical. Why does `throws` come before `throw`?

Solution 4 The code that looks for keywords checks every position in every source line to see if it begins with a keyword. If it finds a match, it overpaints the keyword in the appropriate color. If `throw` came before `throws`, consider what would happen to the following line:

```
void printPaycheck(Employee emp) throws IOException
```

The code would overpaint `throw`, but the `s` would remain black.

Exercise 5 There are several situations in which the project code would improperly draw text in the keyword color. How many of these situations can you name?

Solution 5 The keyword-finding code ignored line comments—that is, comments beginning with a double slash. But it does nothing about comments that begin with slash-star (`/*`) and end with star-slash (`*/`). Any keyword that appeared in such a comment would be overpainted in the keyword color. The following line would look especially strange:

```
/* Let's go forward despite stiff competition. */
```

The "for" in "forward" and the "if" in "stiff" would appear in the keyword color.

Keywords might coincidentally appear in literal strings, for example:

```
System.out.println("while I was dreaming ...");
```

Lastly, keywords might be embedded in the names of classes, variables, or methods:

```
class Republic extends Country { ... }
```

Exercise 6 How would you modify the project code so that `null`, `true`, and `false` are not rendered in the keyword color?

Solution 6 Delete them from the `keywords` array in `FancySrcCanvas`.

Glossary

A

abstract (keyword)

An abstract method has no body. It may not be instantiated. If a class contains any abstract methods, the class itself must be declared abstract.

access modifiers

Keywords that set the access level of classes, data, and methods.

accessors

A method that supports data hiding. It has an empty argument list and returns a data value. By common convention, the name of an accessor method begins with `get`, followed by the property to be retrieved.

additive primary colors

The primary colors of video screens (red, green, and blue). They combine to form yellow, cyan, and magenta.

allocation

Assigning memory for use as objects or arrays.

analog circuit

Non-digital circuit, where precise voltage values are significant.

application

A Java program that's executed in a Java Virtual Machine, consisting of one or more classes.

array

A cluster of variables (components) that are all of the same type. The array has a name, but its individual components do not.

ASCII

An abbreviation for American Standard Code for Information Interchange. ASCII encodes all the characters in American English, plus punctuation marks, into the range 0-127. The range 128-255 encodes symbols such as accented vowels, which are used in western European languages, as well as some Greek characters, line-drawing symbols, and some others.

assembler

Any program that translates assembly code into base-2 instructions.

assembly language

The language of programming with op-codes. Typically, one line of assembly language code corresponds to a single computer instruction.

B

baseline

The imaginary horizontal line on which the bodies of text characters rest.

binary operators

Numeric or boolean operators that take two operands.

bit

The smallest unit of memory, capable of storing 0 or 1. Abbreviation of "binary digit."

Bitwise operation

Operation in which operands are treated as collections of unrelated individual bits. Only performed on integer data types.

block

A contiguous piece of code that begins with an open curly bracket and ends with a matching closed curly bracket.

boolean (keyword)

A primitive data type that represents `true` or `false`.

bounding box

The smallest rectangle that encloses an oval.

byte (keyword)

An 8-bit signed integer primitive data type.

bytecode

The instruction code for the Java Virtual Machine.

C

catch block

Block of code, following a try block, that handles exceptions of a single type.

chaining

The technique of connecting data streams together.

chain of constructors

The mechanism whereby all constructors begin by invoking a constructor of the superclass.

class files

Bytecode output files produced by the compiler.

class loader

Mechanism that finds class files, reads them, and translates them into internal representations.

classpath

A list of directories that contain package structures.

comments

Text used by programmers to help readers understand the meaning of the code.

compiled language

A programming language that must be translated into computed binary. Unlike assembly language, generally a line of source code does not correspond to a single instruction.

component

A GUI device that presents user input to programs and displays program information to users. Standard GUI components include buttons, text fields, scrollbars, and menus. Also: An array member.

conditional code

Code that's executed only when a boolean criterion is satisfied.

construction

Creation of an object or array.

constructor

Code that creates and initializes an instance of a class.

container

A component that can contain other components.

D

data hiding

The practice of making the data of a class as inaccessible to other classes as possible.

debug code

Code whose purpose is to tell the developer about what is going on inside a program.

declaration

Code that tells the compiler the type of a variable or the return type, argument types, and exception types of a method.

default access

Mode that grants access to all classes in the same package as the class that defines the default feature.

default constructor

A no-args constructor created by the compiler for any class that does not have any constructors.

deprecated method

A method that was introduced in an early revision of Java and should not be used.

destination directory

The directory where the compiler will store a package structure.

dialog box

A window, subordinate to its program's main frame, that is used for brief user interaction.

digital circuit

A circuit where voltages represent 0 or 1.

digital computers

Computers composed of digital circuits.

double (keyword)

64-bit floating-point primitive data type.

E

ellipsis

Three dots (...). In a GUI component, an ellipsis indicates that activating the component will cause the display of a new window or dialog.

empty string

An instance of the String class with zero characters.

event-driven program

A program that acts mainly in response to user input.

events

The mechanism by which components inform listeners that they have been activated.

exception

An object that is thrown to indicate an unusual or error state. The throwing of an exception diverts the normal flow of program control.

F

falling through

In `switch` code, continuing from one case to the next in the absence of a `break` statement.

field

A data variable in a class.

file separator

The character that appears between elements in a full pathname.

final (keyword)

A final class may not be subclassed. A final method may not be overridden. A final variable may not be modified after it is initialized.

flag

A `boolean` variable used to indicate program status.

float (keyword)

A 32-bit floating-point primitive data type.

Team LIB

◀ PREVIOUS

NEXT ▶

G

garbage collection

The automatic recycling of unusable objects.

garbage collection thread

The thread that implements garbage collection.

GUI thread

In applications with GUIs, the thread that paints components and notifies event listeners.

Team LIB

◀ PREVIOUS

NEXT ▶

I

immutable

An immutable object's data cannot be changed.

importing

A means to allow the use of abbreviated class names.

index

A unique identifying integer for a component of an array.

inheritance

The mechanism by which a class has the data and methods of its parent classes.

instance variables

Non-static variables of a class.

integer

Any data type that represents non-fractional numbers.

interface

A list of public method declarations.

interpreted compiled language

A language whose compiled code is executed by a virtual machine.

Team LIB

◀ PREVIOUS

NEXT ▶

J

Java Virtual Machine

A virtual computer that runs Java programs.

Team LIB

◀ PREVIOUS

NEXT ▶

L

label

A name associated with a loop. Labels may be used with `break` and `continue` statements.

layout manager

Objects responsible for setting the location and size of components in a container.

listener

An object that should be notified when a component's state changes.

literal string

Text enclosed in double quotes.

look and feel

A GUI-based program's appearance (look) and responses to user input (feel).

loop

A piece of code that's executed repeatedly. The number of repetitions can be preset, or execution can continue until a condition is met.

loop counter

A variable that regulates the number of passes through a loop.

M

maintenance

The process of fixing bugs and adding features.

main thread

The thread that executes an application's `main()` method.

memory

A circuit that stores a digital value.

method caller

The code that calls a method.

modal dialog

A dialog that consumes all mouse and keyboard input.

modulo

An operation that divides the first operand by the second operand and returns the remainder. Its symbol is the `%` sign.

multidimensional

A term used to describe an array with components specified by more than one index.

multithreaded

Capable of performing more than one task at a time.

mutator/setter

Method used to support data hiding. It has a *void* return type and a single argument. By convention, the name of a mutator begins with `set`, followed by the property to be modified.

Team LIB

← PREVIOUS

NEXT →

N

namespace

A way of organizing resources (files, classes, etc.) so that name uniqueness has to be maintained only in relatively small and manageable regions.

nesting

The technique of putting a loop within a loop.

no-args constructor

A constructor with an empty argument list.

Team LIB

← PREVIOUS

NEXT →

O

object equality

An equality criterion that is true if two distinct objects have equal data.

objects

Objects are an individual instance of a class.

one-dimensional

A term used to describe an array with components specified by a single unique index.

operands

The values on which operators operate.

origin

The point with coordinates (0, 0) in a component; the upper-left corner.

overloading

Reuse of a method name in a class.

overriding

Reuse of a method name in an inheritance hierarchy.

P

package

A named group of interrelated classes.

pixel

An abbreviation for *picture element*. A single dot on a computer screen.

precedence

The order of execution when multiple operations are combined into a single statement.

preferred size

The default size of a component, usually derived from its label and font. Layout managers may honor or ignore preferred size.

primitives

The non-object data types: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.

private access

The most restrictive access mode. A private feature may be accessed only by an instance of the class that defines the feature.

protected access

An access mode that grants access to classes in the same package as, and subclasses of, the class that defines the feature.

public access

Completely unrestricted access.

R

radio button

A member of a group, only one of which can be selected at any time.

reader

A class that reads 8-bit text and delivers Unicode characters.

reference

A variable that exists in accessible memory and accesses an object or array in inaccessible memory.

reference equality

An equality criterion that is true if two references point to the same object.

return value

The value returned by a method.

row major

Specification of row followed by column.

S

scalable

Useful and efficient when usage or requirements increase.

scientific notation

A useful representation for expressing very large or very small numbers. The letter E is used as shorthand for "times ten to the...".

scope

A variable's scope is the matching pair of open and closed curly braces that most tightly encloses the variable's declaration.

serifs

Small decorations on the tips of letters that improve readability in medium to large fonts.

shifting

One of several operations that move the bits of an integral operand to the left or right by a certain number of positions.

short-circuit operator

An operator that does not evaluate its second operand if the value of the first operand is enough to determine the value of the operation.

signed

Supporting both positive and negative integer types.

side effect

A change in program state as a result of a method call.

source code

Code that must be translated into appropriate binary values before it can be executed by a computer.

stack trace

A listing of an application's method call hierarchy at the moment an exception was thrown.

static

Associated with a class, rather than with an individual instance of a class.

string concatenation

The consecutive joining of strings, one after another.

subclass

A class that extends a superclass, inheriting its data and methods.

subtractive primary colors

The primary colors of paints and dyes (red, yellow, and blue). They combine to form green, orange, and purple.

superclass

A class from which a subclass inherits data and methods.

T

ternary operator

An operator that takes three operands. Java's only ternary operator is `? : .`

thread

A single task in a multithreaded program.

throw

To interrupt normal program flow by raising an exception.

truncate

To discard the fractional part of a number.

try block

Code following the `try` keyword, from which exceptions might be thrown.

two's complement

A format used to represent signed integers.

Team LIB

← PREVIOUS

NEXT →

U

unary operators

Symbols that perform operations on a single operand.

unicode

A standard for associating characters of many alphabets with 16-bit data.

update

The final part of a for loop.

UTF

A standard for converting Unicode strings into bytes.

Team LIB

← PREVIOUS

NEXT →

Team LIB

◀ PREVIOUS

NEXT ▶

V

variable-width font

Font in which different characters have different widths.

virtual computer

An imaginary computer that is simulated on a real computer.

Team LIB

◀ PREVIOUS

NEXT ▶

W

white space

Blank space in source code, ignored by the compiler but useful in creating code that is more readable.

wrapper

A class whose data is a single primitive value. Java's eight wrapper classes are in the `java.lang` package.

writer

A class that reads Unicode characters and delivers 8-bit text.

Index

Note to the Reader: Throughout this index **boldfaced** page numbers indicate primary discussions of a topic. *Italicized* page numbers indicate illustrations.

A

- abstract classes and methods
 - defined, [468](#)
 - working with, [182–185](#), [183](#)
- access control, [172–174](#)
 - abstract modifier, [182–185](#), [183](#)
 - default access, [174–176](#)
 - final modifier, [180–182](#)
 - with overriding, [178–180](#), [179](#)
 - private access, [174–175](#)
 - protected access, [177–178](#)
 - public access, [174](#)
- access modifiers, [173–174](#), [468](#)
- AccessExample class, [173–174](#)
- accessible memory, [109–111](#), [110](#)
- accessors
 - with data hiding, [173](#)
 - defined, [468](#)
- action listeners, [333–339](#), [335–339](#)
- ActionListener interfaces, [334](#), [363](#)
- actionPerformed method
 - in ActionListener, [334](#)
 - in DisablingNim, [350](#)
 - in FancySrcCanvas, [390–391](#)
 - in FileDialogPractice, [367–368](#)
 - in GraphicOutputNim, [347–348](#)
 - in ListeningFrame, [341](#)
 - in MenuTest, [364–365](#)
 - in Simple Event Lab, [339](#)
 - in SimpleActionListener, [334](#)
 - in SimpleNim, [342–344](#)
 - source of, [339–340](#)
 - in TextAreaNim, [344–345](#)
- add method
 - for border layout managers, [318](#)
 - for buttons, [294](#)
 - for choices, [303](#)
 - for menu items, [307](#), [355](#)
 - for panels, [321](#)
- ADD opcode, [7](#)
- addActionListener method
 - for buttons, [334–335](#)
 - for menus, [363](#)
 - for scrollbars, [355](#)
- addAdjustmentListener method, [355](#)
- addItemListener method
 - for check boxes, [352](#)
 - for choices, [352](#), [376](#)
- addition
 - basic operator for, [37](#)
 - increment operator for, [45–46](#)
- additive primary colors
 - combining, [273–274](#)
 - defined, [468](#)
- addresses of bytes, [4](#), [4](#)
 - in instructions, [16](#)
 - vs. names, [26](#)
 - in opcodes, [7](#)
 - in SimCom, [9–10](#), [9](#)
 - start of, [5](#), [61](#)
 - vs. values, [12](#), [110](#)
- addSeparator method, [307](#)
- addTextListener method, [353](#)
- AdjustmentListener interface, [355](#)

- adjustments, events from, [355–356](#)
- adjustmentValueChanged method, [355–356](#)
- Advanced Exception Lab, [215](#), [216–217](#), [217](#)
- ageInNYears method, [126–127](#), [131](#)
- allocating memory
 - for arrays, [110](#)
 - defined, [468](#)
- American Standard Code for Information Interchange (ASCII)
 - defined, [468](#)
 - for file characters, [263](#)
- ampersands (&)
 - for bitwise operator, [40–41](#)
 - for boolean operator, [46–48](#)
 - in compound assignment, [51](#)
 - as short-circuit operator, [49](#)
- analog circuits
 - defined, [468](#)
 - uses for, [3](#)
- and operators
 - bitwise, [40–41](#)
 - boolean, [46–48](#), [47–48](#)
- AnimatedIllustrations directory, [398](#), [403](#)
- anonymous instances, [289](#)
- api directory, [224](#)
- API pages
 - downloading, [396](#), [403](#)
 - purpose of, [222–223](#)
 - structure of, [224–228](#), [224–227](#)
- append method
 - for text areas, [312](#), [348](#)
 - in TextArea, [344](#)
- Apple Developer Connection site, [401](#)
- applications, [468](#)
- argument bits, [6](#), [6](#)
- arguments
 - command-line, [235–236](#), [235](#)
 - for methods, [60–64](#), [62–63](#)
 - names for, [68](#)
- arithmetic operations
 - basic, [37–38](#)
 - bitwise, [40–41](#), [40–41](#)
 - modulo, [42](#)
 - precedence in, [38–40](#), [39](#)
 - shifting, [42–44](#), [42–44](#)
 - unary, [44–46](#)
- ArrayIndexOutOfBoundsException class, [205–208](#)
- arrays
 - creating, [103–104](#), [103](#)
 - declaring, [102](#)
 - defined, [468](#)
 - exercise questions for, [115–116](#)
 - exercise solutions for, [421–423](#)
 - garbage collection for, [114–115](#)
 - indices for, [102](#), [104](#), [104](#), [470](#)
 - initializing, [103](#), [105](#)
 - length of, [104–105](#)
 - loops for, [105–106](#)
 - multidimensional, [106–108](#), [106](#), [108–109](#)
 - as objects, [109–112](#), [110–112](#)
 - vs. objects, [118–119](#)
 - passing references to, [112–114](#), [113](#)
- ASCII (American Standard Code for Information Interchange)
 - defined, [468](#)
 - for file characters, [263](#)
- assemblers
 - code in, [8](#), [17](#)
 - defined, [468](#)
- assembly languages
 - code in, [8](#), [16](#), [17](#)
 - vs. compiled, [17–18](#), [18](#)
 - defined, [468](#)
- assignment operations, [16](#)
 - compound, [51](#)

operator for, [38](#)
process, [27–28](#)
asterisks (*)
 for comments, [36](#)
 in compound assignment, [51](#)
 with import, [171](#)
 for multiplication, [37](#)
AWT toolkit, [271](#)

Team LIB

◀ PREVIOUS NEXT ▶

Index

B

- background color for frames, [271](#), [275–276](#)
- backslashes (/) in filenames, [249–250](#)
- backward compatibility, [228](#)
- BarAndTF class, [356](#)
- BarAtNorth class, [317](#)
- BarChart class, [183](#), [185](#)
- bars (|)
 - for bitwise operator, [40–41](#)
 - for boolean operator, [46–48](#)
 - in compound assignment, [51](#)
 - for short-circuit operator, [49](#)
- base-2 code, [17](#)
- base-2 notation, [21](#)
- baselines
 - defined, [468](#)
 - for text, [283](#), [283](#)
- batch files, [398](#)
- bin directory, [397](#), [397](#)
- binary operators
 - defined, [468](#)
 - symbols for, [37](#)
- bits
 - defined, [468](#)
 - in memory, [3](#)
 - opcode and argument, [6](#), [6](#)
- bitwise operations
 - defined, [468](#)
 - process, [40–41](#), [40–41](#)
 - right-shift, [43–44](#), [43](#)
- BlackLineOnWhite class, [278](#)
- blocks
 - catch. See [catch blocks](#)
 - defined, [468](#)
 - for if statements, [74](#)
 - scope in, [68](#)
- blue color, [273–274](#)
- BlueRect class, [279](#)
- Boats class, [297–298](#)
- body of methods, [59](#)
- bold font style, [284–285](#)
- BoolArrayLab animated illustration, [108](#), [108–109](#)
- Boolean class, [226](#)
- boolean data type
 - defined, [468](#)
 - for if statements, [74](#)
 - for logical values, [25](#)
 - wrapper class for, [240](#)
- boolean operations
 - comparison, [50–51](#)
 - evaluation of, [46–48](#), [47–48](#)
 - short-circuit, [49–50](#)
- BooleanOps class, [46–47](#)
- BoolLab animated illustration, [47–49](#), [47–48](#)
- border layout managers, [313](#), [317–320](#), [317–319](#)
- bounding boxes
 - defined, [468](#)
 - for ovals, [280–281](#), [281](#)
- BoxLayout layout manager, [325](#)
- break statements
 - labeled, [94–97](#)
 - in loops, [88–89](#)
 - in switch statements, [79–81](#)

- breaking out of loops, [88–89](#)
- BtnInAFrame class, [294](#)
- bugs, finding, [206](#)
- buildColorChoice method, [370](#), [373](#)
- Button class, [293](#)
- buttons
 - in flow layout managers, [316](#)
 - working with, [293–295](#), [293](#), [295](#)
- byte data type
 - defined, [468](#)
 - range of, [21](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- bytecode
 - defined, [468](#)
 - in JVM, [19](#)
- bytes, [3](#), [3](#)
 - addresses of, [4](#), [4](#), [12](#)
 - on disks, [248–249](#)
 - reading, [249](#), [254–255](#), [256](#)
 - writing, [249](#), [251–252](#), [253](#)

Index

C

- callers for methods, [60](#), [471](#)
- calling methods, [60](#), [66–67](#)
- Canvas class, [377](#)
- CardLayout manager, [325](#)
- caret (^)
 - for bitwise operator, [40–41](#)
 - for boolean operator, [46–48](#)
 - in compound assignment, [51](#)
- Cartesian coordinates, [277](#), [278](#)
- case statements, [79](#)
- catch blocks
 - defined, [469](#)
 - execution in, [202–203](#)
 - with instanceof, [212–216](#), [216](#)
 - multiple, [210–212](#)
 - safety net, [213](#)
- catching exceptions, [202–203](#)
- CboxAndChoice class, [352–353](#)
- CboxInnaFrame class, [296](#)
- Center region, [317–319](#), [319](#)
- CenteredOval class, [282](#), [283](#)
- chaining
 - defined, [469](#)
 - input, [259](#), [259](#)
 - output, [256](#), [257](#)
- chains of constructions
 - defined, [469](#)
 - in inheritance, [149–152](#), [149](#)
- char data type
 - with result types, [52–53](#), [53](#)
 - for text, [25](#)
 - wrapper class for, [240](#)
- character code for files, [263](#)
- characters, [25](#)
- charAt method, [233](#)
- charIndexToX method, [383](#), [386](#)
- Chart class, [182–185](#), [183](#)
- checkboxes
 - events from, [351–353](#), [351](#)
 - in flow layout managers, [316](#)
 - working with, [296–300](#), [296–298](#), [300](#)
- CheckboxGroup class, [299](#)
- checked exceptions, [205](#)
 - with stack traces, [216–217](#)
 - throwing, [217–220](#)
 - working with, [208–209](#)
- CheckedCbox class, [297](#)
- Choice class, [302](#)
- choices
 - events from, [351–353](#), [353](#)
 - working with, [301–304](#), [301–302](#), [304](#)
- ChooseFontByRadios class, [301](#)
- circles, [280](#)
- clamp method, [75](#)
- class definitions, [35](#), [120](#)
- class files
 - compiler output, [19](#), [29](#)
 - defined, [469](#)
- class keyword, [119](#)
- class loaders
 - defined, [469](#)

- functions of, [132–133](#), [170](#)
- .class suffix, [19](#), [29](#)
- classes, [119–120](#), [119](#)
 - abstract, [182–185](#), [183](#)
 - core. See [core classes and packages](#)
 - in packages, [165](#)
- classname-dot-staticVariableName syntax, [130](#)
- classpath elements, [168–170](#), [469](#)
- CLASSPATH environment variable
 - for executables
 - in Macintosh, [402–403](#)
 - in Windows, [396](#), [398–399](#)
 - for packages, [169](#)
- classpath option in javac, [169](#)
- close method, [251](#), [258](#)
- closing streams, [251](#)
- colons (:)
 - in classpath elements, [169](#)
 - for labels, [96](#)
 - in ternary operator, [77](#)
- color
 - in final project, [368–376](#), [369](#), [372](#), [378–389](#), [379](#)
 - for frames, [271](#), [275–276](#)
 - for painting, [273–276](#), [275–276](#)
- Color class, [183](#), [237–238](#), [274](#)
- Color Lab program, [275–276](#), [275–276](#)
- ColorChoice class, [374–375](#)
- ColorChoiceTest class, [374–375](#)
- ColorTest class, [372–374](#)
- columns in text areas, [310](#)
- command-line arguments, [235–236](#), [235](#)
- comments
 - defined, [469](#)
 - in Frame Lab, [287](#)
 - painting, [383–384](#)
 - types of, [36](#)
- comparison operators, [50–51](#)
- compatibility, backward, [228](#)
- compiled languages, [16–17](#)
 - vs. assembly, [17–18](#), [18](#)
 - defined, [469](#)
- compiler
 - downloading, [396](#)
 - for packages, [166–167](#), [168](#), [169](#)
 - references with, [155](#)
- compiling, [206](#)
- components, [102](#), [292](#), [292](#)
 - buttons, [293–295](#), [293](#), [295](#)
 - checkboxes
 - events from, [351–353](#), [351](#)
 - in flow layout managers, [316](#)
 - working with, [296–300](#), [296–298](#), [300](#)
 - choices
 - events from, [351–353](#), [353](#)
 - working with, [301–304](#), [301–302](#), [304](#)
 - defined, [469](#)
 - events for. See [events](#)
 - exercise questions for, [327–328](#)
 - exercise solutions for, [448–454](#)
 - labels, [304–305](#), [305](#)
 - layout managers. See [layout managers](#)
 - menus, [305–309](#), [307–308](#)
 - scrollbars, [312–313](#), [313](#)
 - text areas, [310–312](#), [311–312](#)
 - text fields, [309–310](#), [310](#)
- compound assignments, [51](#)
- computePixel method, [90](#)
- concat method, [233](#)
- concatenation of strings, [233](#), [237–239](#), [238–239](#)
- ConcatLab animated illustration, [238–239](#), [238–239](#)

- conditionals, [74](#)
 - defined, [469](#)
 - exercise questions for, [98–99](#)
 - exercise solutions for, [416–420](#)
 - in for loops, [87, 87](#)
 - if statements, [74–76](#)
 - switch statement, [77–81](#)
 - ternary operator, [76–77](#)
- ConnectException class, [210–211, 213–215, 219](#)
- constants
 - benefits of, [181–182](#)
 - in interfaces, [193–194](#)
- construction
 - chains of, [149–152, 149](#)
 - defined, [469](#)
 - with new, [111](#)
- ConstructorLab animated illustration, [150–152](#)
- constructors, [146–147](#)
 - in API pages, [227, 227](#)
 - default, [148–149](#)
 - defined, [469](#)
 - overloading, [147–148](#)
- Container class, [313](#)
- containers, [469](#)
- contexts, graphics, [277](#)
- continue statement
 - labeled, [94–97](#)
 - purpose of, [89–90](#)
- coordinates, [277, 278](#)
- core classes and packages, [205, 222–223](#)
 - API pages for, [222–228, 224–227](#)
 - exercise questions for, [244–246](#)
 - exercise solutions for, [440–442](#)
 - java.lang, [228](#)
 - java.lang.Integer, [240–241](#)
 - java.lang.Math, [243–244](#)
 - java.lang.Object, [236–239, 238–239](#)
 - java.lang.String, [229–236, 231–232, 234–235](#)
 - java.lang.System, [241–243](#)
- cos method, [243](#)
- cp option in javac, [169](#)
- CreateArrayLab animated illusion, [111–113, 113](#)
- .cshrc file for paths, [402](#)
- curly brackets ({})
 - for arrays, [105](#)
 - for constructors, [146](#)
 - for definitions, [35](#)
 - for do-while loops, [86](#)
 - in for loops, [91](#)
 - for if statements, [74](#)
 - for interfaces, [188](#)
 - for method declarations, [59](#)
 - for scope, [68](#)
 - for while loops, [82](#)
- cycloids, [91–92, 91](#)

Index

D

- d option in package, [166–167](#)
- data and data types, [16, 19](#)
 - boolean, [25](#)
 - characters, [25](#)
 - in declarations, [27](#)
 - declaring and assigning, [26–28](#)
 - exercise questions for, [30–31](#)
 - exercise solutions for, [407–409](#)
 - floating-point, [24–25](#)
 - integer, [21–24, 22–23](#)
 - in interfaces, [192–194](#)
 - for objects, [120–122, 121–122](#)
 - summary, [26](#)
- Data Chain Lab animated illustration, [261, 262](#)
- data hiding
 - defined, [469](#)
 - in object-oriented programming, [172–173](#)
- DataInputStream class, [256, 259](#)
- DataLab animated illustration, [122, 122](#)
- DataOutputStream class, [256–257](#)
- debug code, [469](#)
- declarations, [16, 26–28](#)
 - for arrays, [102](#)
 - defined, [469](#)
 - in interfaces, [188–189](#)
 - for methods, [59](#)
- decrement operator, [45–46](#)
- default access
 - defined, [469](#)
 - purpose of, [174–176](#)
- default code, [81](#)
- default constructors
 - defined, [469](#)
 - purpose of, [148–149](#)
- default statements, [79–80](#)
- definitions, class, [35, 120](#)
- deprecated methods
 - compatibility of, [228](#)
 - defined, [469](#)
- destination directories
 - defined, [469](#)
 - for packages, [167](#)
- Developer Tools package, [402](#)
- Developer Tools Update, [401–402](#)
- dialog boxes
 - class for, [365–366, 366](#)
 - defined, [469](#)
- differences, [37](#)
- digital circuits, [2](#)
 - vs. analog, [3](#)
 - defined, [469](#)
- digital computers, [2](#)
 - vs. analog, [3](#)
 - defined, [470](#)
- Dimension class, [282](#)
- dimensions for arrays, [106–108, 106, 108–109](#)
- directories, [165](#)
 - for installation files, [397, 403](#)
 - for packages, [166–167, 166](#)
 - for programs, [398](#)
- disabling components, [349](#)
- DisablingNim class, [349–351](#)
- disks, [248–249](#). *See also* [files](#)

- display method, [183–185](#)
- division
 - operator for, [37](#)
 - truncation with, [40](#)
- do-while loops, [85–86](#)
- docs directory, [224](#)
- Dog class, [120](#)
- double data type
 - defined, [470](#)
 - range of, [24–25](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- double-quotes (") for literal strings, [30](#), [229](#)
- downloading and installing Java
 - in Macintosh, [401–404](#)
 - overview, [396](#)
 - in Windows, [396–401](#), [397](#)
- drawing, [277](#), [278](#). See also [painting](#)
 - circles, [280](#)
 - filling in, [281–282](#), [281](#), [283](#)
 - frames, [287–289](#), [287–289](#)
 - lines, [278](#), [279](#)
 - ovals, [280–281](#), [280–281](#)
 - rectangles, [279](#), [279](#)
 - squares, [280](#)
 - text, [283–286](#), [284–286](#)
- drawLine method, [278](#)
- drawOval method, [280–281](#)
- drawRect method, [279–280](#)
- drawString method, [283–285](#)
- dump method, [129](#)
- dumpSalary method, [146](#)

Index

E

- E variable, [244](#)
- earnMoreThan method, [175](#)
- East region, [317–319](#), [318–319](#)
- Edit menus, guidelines for, [308](#)
- editors for source files, [400](#), [404](#)
- ellipsis (...)
 - for button labels, [336](#)
 - defined, [470](#)
 - for dialog boxes, [309](#)
- else statement, [74–75](#)
- else if statement, [76](#)
- Employee class
 - overriding in, [179](#), [179](#)
 - private access in, [174–175](#)
 - as superclass, [142–143](#), [143](#), [145](#)
- empty strings, [229](#), [470](#)
- EmptyFrame class, [271–272](#)
- environment variable, [169](#)
- equal signs (=)
 - in arithmetic operations, [38](#)
 - for assignment, [27](#)
 - for comparisons, [50](#)
 - for reference equality, [234](#)
- equality
 - object, [234](#), [234](#), [236](#)
 - reference, [234](#), [234](#)
- equals method
 - in Object, [236](#)
 - in Point, [236](#)
 - in String, [233–234](#)
- equalsIgnore method, [233](#)
- error codes and messages, [27](#), [198–200](#)
- escape codes, [28](#)
- EvaluatorLab animated illustration, [39–40](#), [39](#)
- event dispatch threads, [332–333](#)
- event-driven programs, [330–332](#), [331](#)
 - defined, [470](#)
 - threads in, [332–333](#)
- Event Lab animated illustration, [354–355](#), [354](#)
- events, [330](#)
 - actions for, [333–339](#), [335–339](#)
 - from checkboxes, choices, and items, [351–353](#), [351](#), [353](#)
 - defined, [470](#)
 - exercise questions for, [357–358](#)
 - exercise solutions for, [454–460](#)
 - information from, [339–343](#), [340](#), [342](#)
 - from menus, [355](#)
 - in Nim game, [342–351](#), [342–344](#), [346](#), [349](#)
 - from scrollbars and adjustments, [355–356](#), [356](#)
 - from text fields and text areas, [353–355](#), [354–355](#)
- Exception class, [200](#), [205](#), [215](#)
- exceptions, [198](#)
 - catching, [202–203](#)
 - checked, [205](#)
 - with stack traces, [216–217](#)
 - throwing, [217–220](#)
 - working with, [208–209](#)
 - defined, [470](#)
 - exercise questions for, [220](#)
 - exercise solutions for, [438–440](#)
 - families of, [205–206](#)
 - real world, [203](#)
 - runtime, [205–208](#)
 - throwing, [200–201](#), [217–220](#)

- exclamation points (!)
 - for comparisons, [50](#)
 - for inversion, [46–48](#)
- exclusive or operators
 - bitwise, [40–41](#)
 - boolean, [46–48](#), [47–48](#)
- executing bytes, [5](#)
- exit codes, [242](#)
- exit method, [242](#)
- exponents in scientific notation, [24](#)
- expressions in switch statements, [78](#)
- extending interfaces, [194–195](#)
- extends keyword, [142–143](#), [194](#)

Team LIB

← PREVIOUS

NEXT →

Index

F

- falling through switch statements
 - bugs from, [81](#)
 - defined, [470](#)
- false value, [25](#)
- families of fonts, [205–206](#), [284](#)
- FancyButtonInFrame class, [295](#)
- fancysrc package, [389](#)
- FancySrcCanvas class, [377–391](#)
- FancySrcFrame class, [361–362](#), [361](#), [371](#)
- fields
 - in API pages, [227](#), [227](#)
 - defined, [470](#)
 - for objects, [121–122](#)
 - text fields, [309–310](#), [310](#)
 - events from, [353–355](#), [354–355](#)
 - in flow layout managers, [316](#)
- File menu
 - in final project, [362–365](#), [362](#), [364](#)
 - guidelines for, [308](#)
- file separators
 - defined, [470](#)
 - problems with, [249](#)
- FileDialog class, [365–366](#), [366](#)
- FileDialogPractice class, [367–368](#)
- FileInputStream class, [249](#), [254](#)
- filenames, backslashes in, [249–250](#)
- FileNotFoundException class, [251–252](#)
- FileOutputStream class, [249](#), [251](#)
- FileReader class, [263](#), [265](#)
- files
 - character code for, [263](#)
 - Data Chain Lab for, [261](#), [262](#)
 - exercise questions for, [267](#)
 - exercise solutions for, [443–445](#)
 - in final project, [365–368](#), [366](#)
 - line number readers for, [265](#), [266](#)
 - names for, [249–250](#)
 - new lines in, [264–265](#)
 - reading, [249](#)
 - bytes, [254–255](#), [256](#)
 - data, [259–261](#), [259](#), [262](#)
 - as sequences of bytes, [248–249](#)
 - writing, [249](#)
 - bytes, [251–252](#), [253](#)
 - data, [256–259](#), [257](#)
- FileWriter class, [263–264](#)
- Filled class, [281](#)
- filling in drawing, [281–282](#), [281](#), [283](#)
- fillRect method, [281](#)
- final modifier
 - defined, [470](#)
 - working with, [180–182](#)
- final project
 - colors in, [368–376](#), [369](#), [372](#)
 - description of, [360–362](#), [360–361](#)
 - exercise questions for, [392–393](#)
 - exercise solutions for, [461–465](#)
 - File menu in, [362–365](#), [362](#), [364](#)
 - main display area in, [376–378](#), [377](#)
 - painting in, [378–389](#), [379](#)
 - parts of
 - building, [361–362](#)
 - combining, [389–391](#)
 - specifying files in, [365–368](#), [366](#)

- finding packages, [168–170](#)
- Fish class, [177](#)
- flags
 - defined, [470](#)
 - for program status, [95](#)
- float data type
 - defined, [470](#)
 - range of, [24–25](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- floating-point data types, [24–25](#)
- Floating-Point Lab animated illustration, [25](#)
- Flow Lab animated illustration, [316](#), [316](#)
- flow layout managers, [313–316](#), [315–316](#)
- FlowLayout class, [294](#)
- Font class, [285](#)
- Font Lab program, [286](#), [286](#)
- FontAndBaseline class, [283–284](#)
- FontChoice class, [303](#)
- FontChoiceWithLabels class, [304–305](#)
- fonts
 - choices for, [301–304](#), [301–302](#), [304](#)
 - drawing, [283–286](#), [284–286](#)
- for loops
 - for arrays, [105–106](#)
 - structure of, [86–89](#), [87](#)
 - variables in, [97](#)
- fractions, [24](#)
- Frame class, [271](#)
- Frame Lab animated illustration, [287–289](#), [287–289](#)
- frames
 - color for, [271](#), [275–276](#)
 - drawing, [287–289](#), [287–289](#)
 - in painting, [270–273](#), [270](#)
 - text in, [284–285](#), [284–285](#)
- FrameWithSimpleMenu class, [306](#)
- FrameWithSubmenu class, [307–308](#)
- friendly access, [174](#)

Index

G

- garbage collection
 - defined, [470](#)
 - purpose of, [114–115](#)
 - threads for, [332](#)
- garbage collection threads
 - defined, [470](#)
 - purpose of, [332](#)
- getAvailableFontFamilyNames method, [286](#)
- getAverageTemp method, [135–136](#)
- getColorFromChoice method, [371–373](#)
- getDirectory method, [367](#)
- getFile method, [367–368](#)
- getLineNumber method, [265](#)
- getLocalGraphicsEnvironment method, [286](#)
- getMass method, [65–66](#)
- getMessage method, [201–202](#), [219](#)
- getNumEnvelopesInStock method, [210–211](#), [218–220](#)
- getRainfall method
 - error codes for, [198–200](#)
 - exceptions for, [200–203](#)
- getSalary method, [174–175](#)
- getSelectedColor method, [374–375](#)
- getSelectedIndex method, [353](#), [374](#)
- getSize method
 - in Canvas, [380](#)
 - in Frame, [282](#)
- getSource method
 - in ActionEvent, [340](#)
 - in ItemEvent, [352–353](#)
- getState method, [353](#)
- getters, [173](#)
- getValue method, [355](#)
- getWeightKg method, [180–181](#)
- getWeightLbs method, [181](#)
- graphical user interface (GUI)
 - classes for, [228](#)
 - events in. See [events](#)
 - painting in. See [painting](#)
- GraphicOutputNim class, [346–347](#)
- Graphics class, [277–282](#), [278–281](#)
- graphics contexts, [277](#)
- graphics objects, [277](#)
- GraphicsEnvironment class, [286](#)
- greater than signs (>)
 - for comparisons, [50](#)
 - in compound assignment, [51](#)
 - in shifting operations, [42](#)
- green color, [273–274](#)
- GridBagLayout manager, [325](#)
- GridLayout manager, [325](#)
- GUI (graphical user interface)
 - classes for, [228](#)
 - events in. See [events](#)
 - painting in. See [painting](#)
- GUI threads
 - defined, [470](#)
 - in JVM, [332–333](#)

Index

H

HALT opcode, [8](#)

height

of canvas, [380](#)

of dialog boxes, [367](#)

in text areas, [310](#)

Help menus, guidelines for, [309](#)

-help option in java, [236](#)

hiding data, [173](#)

horizontal scrollbars, [312–313](#)

howBig method, [76](#)

Index

I

- IDE (Integrated Development Environment)
 - in Macintosh, [404](#)
 - in Windows, [400](#)
- if statements, [74–76](#)
- immutable classes, [229](#)
- immutable objects, [470](#)
- implements keyword, [188](#)
- import statement, [171–172](#)
- importing
 - defined, [470](#)
 - packages, [170–172](#)
- in variable, [241–242](#)
- inaccessible memory, [109–111](#), [110](#)
- increment operator, [45–46](#)
- incrementing program counter, [7](#)
- indexOf method, [266–267](#), [382](#)
- indices
 - array, [102](#), [104](#), [104](#)
 - defined, [470](#)
- indirect addresses, [7](#)
- information from events, [339–343](#), [340](#), [342](#)
- Inherit Lab animated illustration, [144–145](#), [144–145](#)
- inheritance, [140–142](#)
 - with constructors, [146–152](#), [149](#)
 - defined, [470](#)
 - example, [145–146](#)
 - exercise questions for, [160–161](#)
 - exercise solutions for, [426–431](#)
 - with interfaces, [189–190](#), [189](#)
 - method overriding in, [152–153](#), [152](#)
 - polymorphism with, [154–160](#)
 - from superclasses, [142–145](#), [144–145](#)
- initialization
 - array, [103](#), [105](#)
 - in for loops, [87](#), [87](#), [97](#)
- input, file. See [files](#)
- instance variables, [130](#), [470](#)
- instanceof keyword
 - catch blocks with, [212–216](#), [216](#)
 - for references, [191–192](#)
- instruction sets, [18](#)
- int data type
 - ranges of, [21](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- integers
 - data types for, [21](#)
 - defined, [471](#)
 - two's complement format for, [21–24](#), [22–23](#)
- Integrated Development Environment (IDE)
 - in Macintosh, [404](#)
 - in Windows, [400](#)
- interfaces, [188](#)
 - data in, [192–194](#)
 - defined, [471](#)
 - exercise questions for, [195–196](#)
 - exercise solutions for, [431–435](#)
 - extending, [194–195](#)
 - method declarations in, [188–189](#)
 - objects and references in, [190–192](#)
- interpreted compiled languages
 - defined, [471](#)
 - Java as, [19](#)

- introductory material
 - exercise questions for, [13](#)
 - exercise solutions for, [406–407](#)
 - memory, [2–4](#), [3–4](#)
 - SimCom virtual computer, [5–12](#), [6](#), [9](#)
- inversion operator, [46–48](#)
- invoking methods, [60](#)
- IOException class, [214](#), [218](#), [251–252](#)
- italic font style, [284–285](#)
- item events, [351–353](#)
- ItemListener interface, [352](#)
- itemStateChanged method
 - in CboxAndChoice, [353](#)
 - in ColorChoiceTest, [375](#)
 - in ColorTest, [373–374](#)
 - in FancySrcFrame, [371](#)
 - in ItemListener, [351–352](#)

Index

J

- jar file
 - installing, [397–398](#), [397](#), [400](#)
 - running, [403](#)
- java.awt package, [228](#)
 - API pages for, [225](#)
 - components in, [292](#), [292](#)
- java.awt.Button class, [293](#)
- java.awt.Canvas class, [377](#)
- java.awt.CheckboxGroup class, [299](#)
- java.awt.Choice class, [302](#)
- java.awt.Color class, [183](#), [237–238](#), [274](#)
- java.awt.event.ActionListener interface, [334](#), [363](#)
- java.awt.event.AdjustmentListener interface, [355](#)
- java.awt.event.ItemListener interface, [352](#)
- java.awt.FileDialog class, [365–366](#), [366](#)
- java.awt.Frame class, [271](#)
- java.awt.Graphics class, [277–282](#), [278–281](#)
- java.awt.LayoutManager interface, [325](#)
- java.awt.Panel class, [320](#)
- java.awt.Point class, [236](#)
- java.awt.Scrollbar class, [312](#)
- java.io package, [249](#), [256](#)
- java.lang package, [226](#), [228](#)
- java.lang.Boolean class, [226](#)
- java.lang.Integer class, [240–241](#)
- java.lang.Math class, [243–244](#)
- java.lang.Object class, [236–239](#), [238–239](#)
- java.lang.String class, [226](#), [229–236](#), [231–232](#), [234–235](#)
- java.lang.System class, [241–243](#)
- java.sql package, [228](#)
- java.util package, [228](#)
- Java Virtual Machine (JVM), [12](#), [19](#), [20](#)
 - class loaders in, [132–133](#)
 - defined, [471](#)
 - downloading, [396](#)
 - initialization in, [132](#)
- javac compiler
 - classpath option in, [169](#)
 - directory option in, [167](#)
 - installing, [397–398](#), [397](#)
- javax.swing package, [325](#)
- joining strings, [233](#), [237–239](#), [238–239](#)
- JRE (Java Runtime Environment), [397](#)
- JUMP opcode, [7](#)
- JUMPZ opcode, [7–8](#)
- JVM (Java Virtual Machine), [12](#), [19](#), [20](#)
 - class loaders in, [132–133](#)
 - defined, [471](#)
 - downloading, [396](#)
 - initialization in, [132](#)

Team LIB

← PREVIOUS

NEXT →

Index

K

keywordChoice class, [369](#)

killing frames, [272](#)

Team LIB

← PREVIOUS

NEXT →

Index

L

- L for long data type, [54](#)
- labels
 - defined, [471](#)
 - in flow layout managers, [316](#)
 - for loops, [94–97](#)
 - working with, [304–305](#), [305](#)
- Layout Lab animated illustration, [322–324](#), [322–324](#), [326](#), [326](#)
- layout managers, [294](#), [313–314](#)
 - border, [317–320](#), [317–319](#)
 - CardLayout, GridLayout, and GridBagLayout, [325](#), [326](#)
 - defined, [471](#)
 - flow, [314–316](#), [315–316](#)
 - lab for, [322–324](#), [322–324](#)
 - panels for, [320–324](#), [320](#), [322–324](#)
- LayoutManager interface, [325](#)
- LEFT area, [315](#)
- left-shift operation, [42](#), [42](#)
- length
 - of arrays, [104–105](#)
 - of strings, [233](#)
- length method, [233](#)
- less than signs (<)
 - for comparisons, [50](#)
 - in compound assignment, [51](#)
 - in shifting operations, [42](#)
- license agreements, [397](#), [399](#)
- LineNumberReader class, [265](#), [266](#)
- lines, drawing, [278](#), [279](#)
- listeners
 - defined, [471](#)
 - for events, [333–339](#), [335–339](#)
- ListeningFrame class, [341](#)
- literal strings, [30](#)
 - defined, [471](#)
 - for string instances, [229](#)
- LOAD opcode, [7–8](#)
- logical values, [25](#)
- .login file, [402](#)
- long data type
 - range of, [21](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- look and feel of programs
 - components for. See [components](#)
 - defined, [471](#)
- loop counters, [10](#)
 - defined, [471](#)
 - in for loops, [88](#)
- loops, [10](#), [82](#)
 - for arrays, [105–106](#)
 - breaking out of, [88–89](#)
 - continue statement in, [89–90](#)
 - defined, [471](#)
 - do-while, [85–86](#)
 - exercise questions for, [98–99](#)
 - exercise solutions for, [416–420](#)
 - for, [86–89](#), [87](#)
 - labels for, [94–97](#)
 - nesting, [90–94](#), [91–94](#)
 - scope in, [97](#)
 - while, [82–86](#), [84–85](#)
- lower case characters, [230–232](#), [231–232](#)

Index

M

Macintosh computers, downloading and installing Java on, [401–404](#)

main display area in final projects, [376–378](#), [377](#)

main method, [132–134](#), [133](#)

main threads
defined, [471](#)
purpose of, [332](#)

maintenance
and code duplication, [141](#)
defined, [471](#)

Manager class, [140–141](#), [143](#), [143](#), [151–152](#)

max method, [243](#)

MB prefix, [4](#)

mega prefix, [4](#)

memory, [2–3](#), [3](#)
for arrays, [109–111](#), [110](#)
defined, [471](#)
garbage collection for, [114–115](#)
organization of, [4](#), [4](#)
in SimCom, [6](#), [6](#)

memory leaks, [114](#)

menu bars, [362](#)

Menu class, [307](#)

menuListener, [355](#)

menus
events from, [355](#)
in final project, [362–365](#), [362](#), [364](#)
working with, [305–309](#), [307–308](#)

MenuTest class, [363–365](#)

method callers, [60](#), [471](#)

method definitions, [35](#)

MethodLab animated illustration, [61–64](#), [62](#)

methods, [58](#)
abstract, [182](#)
in API pages, [227](#), [227](#)
arguments for, [60–64](#), [62–63](#)
calling, [60](#), [66–67](#)
deprecated, [228](#)
exercise questions for, [70–71](#)
exercise solutions for, [413–415](#)
final, [180–182](#)
inheritance with, [143](#)
in interfaces, [188–189](#), [194](#)
main, [132–134](#), [133](#)
for objects, [126–127](#), [128](#)
order of execution, [68](#)
overriding, [152–153](#), [152](#)
polymorphism with, [65–66](#), [155–156](#)
references to, [112–114](#), [113](#)
return types for, [60–61](#), [64–65](#)
scope of, [68–69](#)
static, [130–132](#)
structure of, [58–61](#)

min method, [243](#)

minus signs (-)
in compound assignment, [51](#)
for subtraction, [37](#)
as unary operator, [44](#)

modal dialog boxes
characteristics of, [366](#)
defined, [471](#)

modulo operation
defined, [471](#)
operator for, [42](#)

Monospaced fonts, [285](#)

- multi-line comments, [36](#)
- multidimensional arrays
 - defined, [471](#)
 - working with, [106–108](#), [106](#), [108–109](#)
- multiple catch blocks, [210–212](#)
- multiple objects, [122–125](#), [123](#), [125](#)
- multiplication, [37](#)
- multithreaded devices
 - defined, [471](#)
 - JVM as, [332](#)
- mutators
 - with data hiding, [173](#)
 - defined, [471](#)

Team LIB

◀ PREVIOUS NEXT ▶

Index

N

- `\n` character, [264–265](#)
- n-dimensional arrays, [107](#)
- names
 - for arguments, [68](#)
 - for classes, [165](#)
 - for constructors, [146](#)
 - in declarations, [27](#)
 - for files, [249–250](#)
 - for memory locations, [26](#)
 - for methods, [60](#)
 - reusing, [154](#)
 - for variables, [68–69](#)
- namespaces
 - defined, [471](#)
 - directories for, [165](#), [165](#)
- nCubed method, [67](#)
- NEAndW class, [319](#)
- negative numbers, [21](#), [23](#)
- NestedLoopLab animated illustration, [91–94](#), [92–94](#)
- nesting
 - defined, [472](#)
 - if statements, [75](#)
 - loops, [90–94](#), [91–94](#)
 - menus, [308](#)
 - parentheses, [39](#)
- new keyword, [103](#)
- newline characters
 - in files, [264–265](#)
 - printing, [242](#)
 - for text areas, [312](#)
- Nim game, [342–351](#), [342–344](#), [346](#), [349](#)
- Nim Lab program, [343–351](#), [343–344](#), [346](#), [349](#)
- no-args constructors, [148](#)
 - defined, [472](#)
 - with superclasses, [150](#)
- NoMethods class, [58](#)
- North region, [317–319](#), [318–319](#)
- null value
 - with readline, [266](#)
 - with references, [112](#), [135–136](#)
- NullLayout class, [326](#)
- NumberFormatException class, [241](#)
- numeric operations, [52](#)

Index

O

- Object class, [236–239](#), [238–239](#)
- object-oriented programming, [119](#)
- ObjectLifeCycleLab animated illustration, [133–134](#), [133–134](#)
- ObjectMethodLab animated illustration, [127](#), [127](#)
- objects, [118](#)
 - vs. arrays, [118–119](#)
 - arrays as, [109–112](#), [110–112](#)
 - classes, [119–120](#), [119](#)
 - data for, [120–122](#), [121–122](#)
 - defined, [472](#)
 - equality of, [234](#), [234](#), [236](#), [472](#)
 - exercise questions for, [136–137](#)
 - exercise solutions for, [423–426](#)
 - methods for, [126–127](#), [128](#)
 - multiple, [122–125](#), [123](#), [125](#)
 - reference data with, [134–136](#)
 - references to, [121](#), [121](#), [190–192](#)
 - static data in, [128–130](#)
 - static methods in, [130–132](#)
- odometers, base-2, [21–22](#), [22](#)
- Officer class, [151–153](#)
- one-dimensional arrays
 - characteristics of, [106](#)
 - defined, [472](#)
- opcode bits, [6](#), [6](#)
- opcodes, [6–7](#)
- Open... menu item, [362](#)
- operands, [37](#), [472](#)
- operation codes, [6](#), [6](#)
- operations, [34](#)
 - arithmetic
 - basic, [37–38](#)
 - bitwise, [40–41](#), [40–41](#)
 - modulo, [42](#)
 - precedence in, [38–40](#), [39](#)
 - shifting, [42–44](#), [42–44](#)
 - unary, [44–46](#)
 - boolean
 - comparison, [50–51](#)
 - evaluation in, [46–48](#), [47–48](#)
 - short-circuit, [49–50](#)
 - comments, [36](#)
 - compound assignment, [51](#)
 - exercise questions for, [55–56](#)
 - exercise solutions for, [409–412](#)
 - result types in, [52–54](#), [53](#)
 - white space, [34](#)
- or operators
 - bitwise, [40–41](#)
 - boolean, [46–48](#), [47–48](#)
- order of method execution, [68](#)
- origins
 - defined, [472](#)
 - in drawing, [277](#), [278](#)
- OS X Developer Tools, [401](#)
- out variable, [241–242](#)
- output
 - file. See [files](#)
 - printing, [29–30](#)
- Oval class, [133](#)
- ovals
 - drawing, [280–281](#), [280–281](#)
 - filled, [281–282](#), [281](#)
- OverlayLayout layout managers, [325](#)

overloading
 constructors, [147–148](#)
 defined, [472](#)
 methods, [65](#)

overriding
 access control with, [178–180](#), [179](#)
 defined, [472](#)
 methods, [152–153](#), [152](#)

Team LIB

◀ PREVIOUS NEXT ▶

Index

P

- package access, [174](#)
- package keyword, [166–167](#)
- packages, [164](#), [164](#)
 - access control in. See [access control](#)
 - core. See [core classes and packages](#)
 - creating, [166–167](#), [166](#), [168](#)
 - defined, [472](#)
 - exercise questions for, [186](#)
 - exercise solutions for, [435–438](#)
 - finding, [168–170](#)
 - importing, [170–172](#)
 - interfaces in, [188](#)
 - and namespaces, [165–166](#), [165](#)
- paint method, [276–277](#)
 - in BlackLineOnWhite, [278](#)
 - in BlueRect, [279](#)
 - in CenteredOval, [282](#)
 - in ColorChoiceTest, [376](#)
 - in ColorTest, [373–374](#)
 - in DisablingNim, [350](#)
 - in FancySrcCanvas, [378](#), [380](#), [387](#)
 - in Filled, [281](#)
 - in FontAndBaseline, [284](#)
 - in Frame, [287](#)
 - in GraphicOutputNim, [347–348](#)
 - in ThreeOvals, [280](#)
 - in Xxx, [331](#)
- painting, [270](#)
 - color for, [273–276](#), [275–276](#)
 - drawing shapes. See [drawing](#)
 - exercise questions for, [290](#)
 - exercise solutions for, [445–448](#)
 - in final project, [378–389](#), [379](#)
 - Frame Lab for, [287–289](#), [287–289](#)
 - frames in, [270–273](#), [270](#)
 - process, [276–277](#)
 - text, [283–286](#), [284–286](#)
- paintLines method, [380](#), [387](#)
- paintOneSourceLine method, [382–384](#), [388–389](#)
- paintRegion method
 - in BarChart, [185](#)
 - in Chart, [185](#)
 - in PieChart, [185](#)
- paintText method, [380–382](#), [387–388](#)
- Panel class, [320](#)
- PanelInFrame class, [320–321](#)
- panels, [320–324](#), [320](#), [322–324](#)
- parabolas, [108](#), [109](#)
- parentheses ()
 - in arithmetic operations, [38–40](#), [39](#)
 - in boolean operations, [47](#)
 - in do-while loops, [86](#)
 - in for loops, [87](#)
 - for if statements, [74](#)
 - for methods, [60](#)
- parseInt method, [241](#)
- PartTimer class, [180](#)
- PassArrayLab animated illustration, [113–114](#)
- passing
 - arguments, [63](#), [67–68](#)
 - references to methods, [112–114](#), [113](#)
- PATH environment variable, [396](#), [398–399](#), [403](#)
- payEveryone method, [156–159](#)
- Paymaster class, [156–159](#), [180](#)
- percent signs (%)

- in compound assignment, [51](#)
 - in modulo operations, [42](#)
- periods (.) for object properties, [118](#)
- Person class, [120–123](#), [126–131](#)
- PI variable, [244](#)
- PieChart class, [183](#), [185](#)
- pixels
 - defined, [472](#)
 - for frames, [90](#), [272](#)
- plain font style, [284–285](#)
- plus signs (+)
 - for addition, [37](#)
 - in compound assignment, [51](#)
 - for string concatenation, [237–238](#)
 - as unary operator, [44](#)
- PlusPlusMinusMinus class, [45](#)
- Point class, [236](#)
- Point3D class, [239](#), [239](#)
- polymorphism
 - with inheritance, [154–160](#)
 - with methods, [65–66](#)
- position
 - in drawings, [277](#), [278](#)
 - of text, [283](#)
- post-decrement operator, [46](#)
- post-increment operator, [46](#)
- PostDec class, [46](#)
- pow method, [243](#)
- pre-decrement operator, [46](#)
- pre-increment operator, [46](#)
- precedence
 - in arithmetic operations, [38–40](#), [39](#)
 - in boolean operations, [47](#)
 - defined, [472](#)
 - summary, [54–55](#)
- preferred size
 - defined, [472](#)
 - with layout managers, [314](#), [317](#)
- primary colors, [273–274](#)
- primitive data types
 - defined, [472](#)
 - summary, [26](#)
- print2Cubes method, [66–67](#)
- print2Vals method, [66](#)
- print3x method, [67](#)
- printChars method, [233](#)
- printCheck method
 - in Employee, [142–143](#), [145–146](#), [179](#)
 - in Officer, [153](#)
 - overriding, [152–153](#)
 - in PartTimer, [180](#)
 - in Worker, [140–141](#)
- PrinterIOException class, [205](#), [209](#), [211](#), [213–215](#)
- printHelpMessage method, [235](#)
- printing, [30](#)
- println method, [60](#), [242](#), [364](#)
- printPretty method, [64](#)
- printRetAddr method, [208–211](#)
- printSomeEnvelopes method, [209–213](#)
- printStackTrace method, [216–217](#)
- printTriple method, [69](#)
- printWeight method, [177–178](#)
- private access
 - defined, [472](#)
 - working with, [173–175](#)
- products, [37](#)
- program counters
 - incrementing, [7](#)

- purpose of, [5](#)
- program files
 - for Macintosh installation, [404](#)
 - for Windows installation, [400–401](#)
- properties of objects, [118](#)
- protected access, [173](#)
 - defined, [472](#)
 - working with, [177–178](#)
- public access
 - defined, [472](#)
 - working with, [173–174](#)

Team LIB

4 PREVIOUS NEXT 5

Team LIB

← PREVIOUS

NEXT →

Index

Q

QTJava, [401](#)

question marks (?) in ternary operator, [ZZ](#)

quotients, [3Z](#)

Team LIB

← PREVIOUS

NEXT →

Index

R

- ␣ character, [264–265](#)
- radio buttons
 - defined, [472](#)
 - working with, [299–300](#), [300](#)
- RadioBoats class, [299–300](#)
- RAM, [248–249](#). See also [memory](#)
- random method, [243](#)
- RandomAreas class, [244](#)
- read method, [254](#)
- Read10Bytes class, [254–255](#)
- readBoolean method, [260](#)
- readByte method, [260](#)
- Reader class, [263](#)
- readers, [263](#), [264](#), [472](#)
- reading, [249](#)
 - bytes, [254–255](#), [256](#)
 - data, [259–261](#), [259](#), [262](#)
- readLine method, [265–266](#)
- readUTF method, [260](#), [263](#)
- ReadWithChain class, [260](#)
- reconfigure method, [377–378](#), [386–387](#)
- Rectangle class, [133](#)
- rectangle method, [281](#)
- rectangles
 - drawing, [279](#), [279](#)
 - filled, [281](#), [281](#)
- red color, [273–274](#)
- redrawing, [276](#)
- reference-dot notation, [126](#)
- reference variables, [109](#)
- references
 - to arrays, [110–112](#), [112](#)
 - with compiler, [155](#)
 - defined, [472](#)
 - equality of, [234](#), [234](#), [472](#)
 - to methods, [112–114](#), [113](#)
 - to objects, [121](#), [121](#), [190–192](#)
 - passing arguments by, [67](#)
 - for variables, [134–136](#)
- regions in layout managers, [317–319](#), [318–319](#)
- registers, [5](#), [6](#)
- RemoteEnvelopeCountException class, [219](#)
- removeActionListener method, [335](#)
- removeItemListener method, [352](#)
- repaint method
 - for ColorTest, [374](#)
 - for main display area, [377–378](#)
 - in Nim game, [348](#)
- result types in operations, [52–54](#), [53](#)
- return character, [264–265](#)
- return statement, [64](#)
- return types and values
 - defined, [472](#)
 - for methods, [60–61](#), [64–65](#)
- ReusesNames class, [69](#)
- RIGHT area, [315](#)
- right-shift operation, [42–44](#), [43](#)
- row major order
 - defined, [472](#)

in text areas, [310](#)
rows in text areas, [310](#)
Run Lightspeed option, [10](#)
runtime exceptions, [205–208](#)
RuntimeException class, [205](#)

Team LIB

◀ PREVIOUS

NEXT ▶

Index

S

- safety net catch blocks, [213](#)
- Sans Serif fonts, [285–286](#)
- scalability
 - defined, [473](#)
 - of events, [330](#)
- scientific notation
 - defined, [473](#)
 - purpose of, [24](#)
- scope
 - defined, [473](#)
 - in loops, [97](#)
 - of methods, [68–69](#)
- scripts, [398](#)
- Scrollbar class, [312](#)
- scrollbars
 - creating, [312–313](#), [313](#)
 - events from, [355–356](#), [356](#)
 - in text areas, [310–312](#), [311–312](#)
- SDK (Software Development Kit), [396–397](#)
- semicolons (;)
 - in classpath elements, [169](#)
 - in declarations, [27](#)
 - in do-while loops, [86](#)
 - in for loops, [87](#), [91](#)
- Serif fonts, [285–286](#)
- serifs, [473](#)
- setBackground method
 - in Button, [294](#)
 - for frames, [271](#), [275–276](#)
- setBounds method, [325](#)
- setColor method, [277](#)
- setColorScheme method, [183–184](#)
- setEnabled method, [349](#)
- setFont method, [284–285](#), [294](#)
- setForeground method, [294](#)
- setLayout method, [313–314](#), [325](#)
- setLocation method, [325](#)
- setNumEnvelopesInStock method, [210–211](#)
- setSalary method, [175](#)
- setSize method, [272](#), [282](#), [325](#), [367](#)
- setters
 - with data hiding, [173](#)
 - defined, [471](#)
- setTitle method, [271](#)
- setValues method, [183–184](#)
- setVisible method
 - for dialog boxes, [367](#)
 - for frames, [272–273](#), [289](#), [333](#)
- SeveralObjectsLab animated illustration, [123–125](#), [125](#)
- shifting operations
 - defined, [473](#)
 - process, [42–44](#), [42–44](#)
- ShiftLab animated illustration, [43–44](#), [44](#)
- short-circuit operators
 - in Boolean operations, [49–50](#)
 - defined, [473](#)
- short data type
 - range of, [21](#)
 - with result types, [52–53](#), [53](#)
 - wrapper class for, [240](#)
- ShowButton class, [293](#)

- ShowMeATrace class, [207](#)
- side effects
 - defined, [473](#)
 - with methods, [64–65](#)
- signed integer types
 - defined, [473](#)
 - summary, [21](#)
- SimCom computer, [5–8](#), [6](#)
 - benefits of, [11–12](#)
 - working with, [8–11](#), [9](#)
- Simple Base 2 animated illustration, [21–24](#), [22–23](#)
- Simple Event Lab animated illustration, [336–339](#), [336–339](#)
- Simple Exception Lab animated illustration, [203](#), [204](#)
- Simple Input Lab animated illustration, [255](#), [256](#)
- Simple Output Lab animated illustration, [252](#), [253](#)
- SimpleActionListener class, [334](#)
- SimpleChoice class, [302](#)
- SimpleFlow class, [314–315](#)
- SimpleNim class, [342–351](#), [342–344](#), [346](#), [349](#)
- sin method, [243](#)
- single-line comments, [36](#)
- size
 - of arrays, [103–105](#)
 - of canvas, [380](#)
 - of dialog boxes, [367](#)
 - of fonts, [284](#)
 - of frames, [272](#)
 - with layout managers, [314](#), [317](#)
 - of strings, [233](#)
- slashes (/)
 - for comments, [36](#)
 - in compound assignment, [51](#)
 - for division, [37](#)
- Software Development Kit (SDK), [396–397](#)
- source code, [16](#)
 - creating
 - in Macintosh, [404](#)
 - in Windows, [400–401](#)
 - defined, [473](#)
- South region, [317–319](#)
- specifying files in final project, [365–368](#), [366](#)
- SpringLayout layout manager, [325](#)
- square brackets ([]) for arrays, [102](#)
- squares, [280](#)
- stack traces
 - checked exceptions with, [216–217](#)
 - defined, [473](#)
 - runtime exceptions with, [206–208](#)
- startsWith method, [233](#), [385](#)
- statements
 - declarations, [27](#)
 - in for loops, [91](#)
- static classes, [473](#)
- static data, [128–130](#)
- static methods, [126–127](#), [130–132](#)
- static modifier, [60](#)
- Step Lightspeed option, [10](#)
- STORE opcode, [7](#)
- storeCubes method, [207–208](#)
- storeOneCube method, [207–208](#)
- String class, [188](#)
 - API pages for, [226](#)
 - for command-line arguments, [235–236](#), [235](#)
 - working with, [229–234](#), [231–232](#), [234](#)
- string concatenation, [233](#), [237–238](#)
 - defined, [473](#)
 - lab for, [238–239](#), [238–239](#)
- StringLab animated illustration, [231–232](#), [231–232](#)

- styles of fonts, [284–285](#)
- SUB opcode, [7](#)
- subclasses
 - defined, [473](#)
 - inheritance by, [140–142](#)
 - protected access with, [177](#)
- Submarine class, [149–150](#), [149](#)
- subpackages, [165](#)
- substring method, [233](#), [382](#), [385](#)
- subtraction
 - basic operator for, [37](#)
 - decrement operator for, [45–46](#)
- subtractive primary colors, [273](#), [473](#)
- sums, [37](#)
- super keyword, [151–152](#)
- supercategories, [142](#)
- superclasses, [140–142](#)
 - defined, [473](#)
 - inheritance from, [142–145](#), [144–145](#)
- Swing toolkit
 - for GUI, [271](#)
 - for layout managers, [325](#)
- switch statements, [77–79](#)
 - break statements in, [79–81](#)
 - default statements in, [79–80](#)
- System class, [241–243](#)
- System.exit call, [333](#), [343](#)

Index

T

- TAlnnaFrame class, [311](#)
- Talker interface, [188](#)
- tan method, [243](#)
- ternary operator
 - defined, [473](#)
 - operation of, [76–77](#)
- text
 - drawing, [283–286](#), [284–286](#)
 - in final project, [379](#), [379](#)
- text areas
 - events from, [353–355](#), [354–355](#)
 - working with, [310–312](#), [311–312](#)
- text characters, [25](#)
- text fields
 - events from, [353–355](#), [354–355](#)
 - in flow layout managers, [316](#)
 - working with, [309–310](#), [310](#)
- text files, [263](#)
- text listeners, [353](#)
- TextAreaNim class, [344–345](#)
- TextField constructor, [309](#)
- TextListener interface, [353](#)
- textValueChanged method, [353](#)
- TFs class, [309–310](#)
- Thermometer class, [135](#)
- this keyword, [131](#)
- this-reference notation, [131](#)
- threads
 - defined, [473](#)
 - in event-driven programs, [332–333](#)
- ThreeOvals class, [280](#)
- throw keyword, [200](#), [473](#)
- throwing exceptions
 - checked exceptions, [217–220](#)
 - process, [200–201](#)
- throws keyword, [200](#)
- tildes (~) in bitwise operations, [40](#), [40](#)
- toLowerCase method, [230–232](#), [231–232](#)
- toString method, [226](#)
 - in Object, [236–238](#)
 - in String, [239](#)
- toThe5th method, [59–61](#)
- toUpperCase method, [230–232](#), [231–232](#)
- traditional comments, [36](#)
- Transport class, [149–150](#), [149](#)
- Triangle class, [133](#)
- trim method, [233](#)
- trinary operator, [37](#)
- true value, [25](#)
- truncation
 - defined, [473](#)
 - with division, [40](#)
- try blocks, [205](#)
 - defined, [473](#)
 - working with, [202–203](#)
- Tuna class, [177–178](#)
- two-dimensional arrays, [106–107](#), [106](#)
- TwoBars class, [312–313](#)
- two's complement format

defined, [473](#)
for integer types, [21-23](#), [22](#)

Team LIB

← PREVIOUS

NEXT →

Index

U

- unary operators, [37](#)
 - arithmetic, [44–46](#)
 - defined, [473](#)
- Unicode standard, [25](#)
 - defined, [474](#)
 - for files, [257, 263](#)
- unnamed packages, [176](#)
- updates
 - defined, [474](#)
 - in for loops, [87, 87](#)
- upper case characters, [230–232, 231–232](#)
- useColor method, [183–184](#)
- UsesListener class, [334–335](#)
- UsesMethods class, [59](#)
- UTF standard
 - defined, [474](#)
 - for files, [257, 263](#)

Index

V

- values
 - vs. addresses, [12](#), [110](#)
 - logical, [25](#)
 - in memory, [110](#)
 - passing arguments by, [67–68](#)
- variable-width fonts
 - defined, [474](#)
 - vs. monospaced, [285](#)
- variables, [26](#)
 - for array size, [103](#)
 - for color, [274](#)
 - declaring, [27](#)
 - final, [180](#)
 - in for loops, [97](#)
 - inheritance with, [143](#)
 - instance, [130](#), [470](#)
 - for objects, [121–122](#)
 - polymorphism with, [155–156](#)
 - references for, [134–136](#)
 - scope of, [68–69](#)
- verbose option in java, [236](#)
- version option in java, [403](#)
- vertical bars (|)
 - for bitwise operator, [40–41](#)
 - for boolean operator, [46–48](#)
 - in compound assignment, [51](#)
 - for short-circuit operator, [49](#)
- vertical scrollbars, [312–313](#)
- VerySimple class, [28–29](#), [35](#)
- VerySimple2 class, [29–30](#)
- virtual computers, [19](#)
 - defined, [474](#)
 - JVM. See [JVM \(Java Virtual Machine\)](#)
 - SimCom, [12](#)
- visibility
 - of dialog boxes, [367](#)
 - of frames, [272–273](#)
- vocalHypotSquared method, [65](#)
- voltages, [2](#)

Index

W

- WaterTransport class, [149–150](#), [149](#)
- WeatherStation class, [135](#)
- West region, [317–319](#), [318–319](#)
- while loops, [82–86](#), [84–85](#)
- WhileLab animated illustration, [83–86](#), [84–85](#)
- white space
 - defined, [474](#)
 - in program code, [34–36](#)
- width
 - of canvas, [380](#)
 - of dialog boxes, [367](#)
 - with result types, [52–54](#), [53](#)
 - in text areas, [310](#)
- Windows computers, downloading and installing Java on, [396–401](#), [397](#)
- Worker class, [140–141](#)
 - constructors for, [146–147](#)
 - as subclass, [143](#), [143](#), [145–146](#)
- wrapper classes, [240](#)
 - benefits of, [241](#)
 - defined, [474](#)
- write method, [251](#)
- Write10Bytes class, [251–252](#)
- writeByte method, [257](#)
- writeChar method, [257](#)
- Writer class, [263](#)
- writers, [263](#), [264](#), [474](#)
- writeShort method, [257](#)
- writeUTF method, [257](#), [263](#)
- WriteWithChain class, [258](#)
- writing, [249](#)
 - bytes, [251–252](#), [253](#)
 - data, [256–259](#), [257](#)

Team LIB

← PREVIOUS

NEXT →

Index

X

X39 class, [241](#)

X39RevB class, [242–243](#)

Team LIB

← PREVIOUS

NEXT →

Team LIB

PREVIOUS NEXT

Index

Z

Zebra class, [180–182](#)

zeroth array components, [104](#)

zip files, [399](#)

Team LIB

PREVIOUS NEXT

List of Figures

Chapter 1: An Introduction to Computers That Will Actually Help You in Life

[Figure 1.1](#): A bit

[Figure 1.2](#): A byte

[Figure 1.3](#): Several bytes

[Figure 1.4](#): SimCom architecture

[Figure 1.5](#): Opcode and argument bits

[Figure 1.6](#): SimCom in action

Chapter 2: Data

[Figure 2.1](#): Assembly language

[Figure 2.2](#): Compiled language

[Figure 2.3](#): Evolution of a Java application

[Figure 2.4](#): SimpleBase2Lab

[Figure 2.5](#): A base-2 odometer

[Figure 2.6](#): An example of two's complement

[Figure 2.7](#): Two's complement lab

Chapter 3: Operations

[Figure 3.1](#): EvaluatorLab

[Figure 3.2](#): EvaluatorLab after evaluation

[Figure 3.3](#): The unary bitwise operator ~

[Figure 3.4](#): Bitwise "and"

[Figure 3.5](#): Left-shift: <<

[Figure 3.6](#): Bitwise right-shift: >>>

[Figure 3.7](#): Numeric right-shift: >>

[Figure 3.8](#): ShiftLab

[Figure 3.9](#): ShiftLab after shifting

[Figure 3.10](#): BoolLab: initial screen

[Figure 3.11](#): BoolLab after execution

[Figure 3.12](#): Data type width, not to scale

[Figure 3.13](#): Data type width relationships

Chapter 4: Methods

[Figure 4.1](#): MethodLab

[Figure 4.2](#): MethodLab after animating

[Figure 4.3](#): Numeric type widths

Chapter 5: Conditionals and Loops

[Figure 5.1](#): While Lab: initial display

[Figure 5.2](#): While Lab with modified test expression

[Figure 5.3](#): While Lab after execution

[Figure 5.4](#): A common loop usage

[Figure 5.5](#): A cycloid

[Figure 5.6:](#) NestedLoopLab: initial display

[Figure 5.7:](#) NestedLoopLab: 8:15

[Figure 5.8:](#) NestedLoopLab with a loop

[Figure 5.9:](#) NestedLoopLab with nested loops

Chapter 6: Arrays

[Figure 6.1:](#) A new array

[Figure 6.2:](#) A used array

[Figure 6.3:](#) A two-dimensional array

[Figure 6.4:](#) BoolArrayLab

[Figure 6.5:](#) BoolArrayLab drawing a parabola

[Figure 6.6:](#) Accessible and inaccessible memory

[Figure 6.7:](#) An array of bytes in inaccessible memory

[Figure 6.8:](#) Reference and array

[Figure 6.9:](#) Two references, one array

[Figure 6.10:](#) CreateArrayLab

Chapter 7: Introduction to Objects

[Figure 7.1:](#) Class as mental category

[Figure 7.2:](#) Reference and object

[Figure 7.3:](#) DataLab

[Figure 7.4:](#) Multiple objects

[Figure 7.5:](#) SeveralObjectsLab

[Figure 7.6:](#) SeveralObjectsLab reconfigured

[Figure 7.7:](#) SeveralObjectsLab reconfigured and executed

[Figure 7.8:](#) ObjectMethodLab

[Figure 7.9:](#) ObjectLifeCycleLab

[Figure 7.10:](#) ObjectLifeCycleLab after running a while

Chapter 8: Inheritance

[Figure 8.1:](#) A Simple inheritance hierarchy

[Figure 8.2:](#) Inherit Lab

[Figure 8.3:](#) Inherit Lab's class-editing dialog box

[Figure 8.4:](#) Object layers

[Figure 8.5:](#) Inheritance of `Officer`

Chapter 9: Packages and Access

[Figure 9.1:](#) Example package/ directory structure

[Figure 9.2:](#) Package as namespace

[Figure 9.3:](#) Initial directory structure

[Figure 9.4:](#) After compilation

[Figure 9.5:](#) After more compilation

[Figure 9.6:](#) Polymorphism revisited

[Figure 9.7:](#) Chart class and subclasses

Chapter 10: Interfaces

[Figure 10.1](#): Animal kingdom class inheritance

Chapter 11: Exceptions

[Figure 11.1](#): Simple Exception Lab

[Figure 11.2](#): Simple Exception Lab: final state with normal execution

[Figure 11.3](#): Advanced Exception Lab

[Figure 11.4](#): Choosing an exception type in Advanced Exception Lab

[Figure 11.5](#): Advanced Exception Lab reconfigured

Chapter 12: The Core Java Packages and Classes

[Figure 12.1](#): Structure of the API index

[Figure 12.2](#): Structure of the classes frame

[Figure 12.3](#): Class description

[Figure 12.4](#): Field/constructor/ method summaries

[Figure 12.5](#): StringLab

[Figure 12.6](#): StringLab: uppercase, 2 references

[Figure 12.7](#): StringLab: lowercase, 1 reference

[Figure 12.8](#): String references and objects

[Figure 12.9](#): Command-line arguments

[Figure 12.10](#): ConcatLab

[Figure 12.11](#): ConcatLab's Point3D class

[Figure 12.12](#): ConcatLab's Point3D class

Chapter 13: File Input and Output

[Figure 13.1](#): Simple Output Lab

[Figure 13.2](#): Simple Output Lab in progress

[Figure 13.3](#): Simple Input Lab in progress

[Figure 13.4](#): Output chaining

[Figure 13.5](#): Input chaining

[Figure 13.6](#): Data Chain Lab

[Figure 13.7](#): Data Chain Lab in progress: Text, writers, and readers

[Figure 13.8](#): Readers and writers

[Figure 13.9](#): Line number reader and file reader

Chapter 14: Painting

[Figure 14.1](#): A frame with boring contents

[Figure 14.2](#): Color Lab

[Figure 14.3](#): Color Lab with a predefined color

[Figure 14.4](#): Pixel coordinates

[Figure 14.5](#): A black line on a white background

[Figure 14.6](#): A rectangle

[Figure 14.7](#): Ovals and bounding boxes

[Figure 14.8](#): Three ovals

[Figure 14.9](#): Filled rectangle and ovals

[Figure 14.10](#): Original CenteredOval

[Figure 14.11](#): Resized CenteredOval

[Figure 14.12](#): The baseline

[Figure 14.13](#): Text and baseline in a frame

[Figure 14.14](#): Text in a frame

[Figure 14.15](#): Font Lab

[Figure 14.16](#): Font Lab with an exotic font

[Figure 14.17](#): Initial Frame Lab display

[Figure 14.18](#): Frame Lab with custom configuration

[Figure 14.19](#): The result of Figure 14.18

Chapter 15: Components

[Figure 15.1](#): A component sampler

[Figure 15.2](#): A button in a frame

[Figure 15.3](#): A fancy button

[Figure 15.4](#): A simple checkbox

[Figure 15.5](#): A checked checkbox

[Figure 15.6](#): Three checkboxes and a button

[Figure 15.7](#): Checkboxes as radio buttons

[Figure 15.8](#): Multiple checkbox groups

[Figure 15.9](#): A choice

[Figure 15.10](#): An expanded choice

[Figure 15.11](#): Two choices

[Figure 15.12](#): Choices with labels

[Figure 15.13](#): A menu in a menu bar

[Figure 15.14](#): A menu with a separator

[Figure 15.15](#): Hierarchical menus

[Figure 15.16](#): Two text fields

[Figure 15.17](#): A text area

[Figure 15.18](#): Multiple checkbox groups

[Figure 15.19](#): A text area with scroll bars

[Figure 15.20](#): A pair of disappointing scrollbars

[Figure 15.21](#): Flow layout manager

[Figure 15.22](#): Wider

[Figure 15.23](#): Narrower

[Figure 15.24](#): Left-aligned

[Figure 15.25](#): Flow Lab

[Figure 15.26](#): Scrollbar at North

[Figure 15.27](#): North and South occupied

[Figure 15.28](#): North, East, and West occupied

[Figure 15.29](#): North, East, West, and Center occupied

[Figure 15.30](#): A panel in a frame

[Figure 15.31](#): Layout lab

[Figure 15.32](#): Layout lab's frame editing dialog

[Figure 15.33](#): Layout Lab with an added panel

[Figure 15.34](#): A button in a panel in a frame

[Figure 15.35](#): Layout Lab makes it so

[Figure 15.36](#): No layout manager

Chapter 16: Events

- [Figure 16.1](#): A GUI waiting for events
- [Figure 16.2](#): A button that sends events
- [Figure 16.3](#): Simple Event Lab: initial screen
- [Figure 16.4](#): Simple Event Lab with simulated buttons
- [Figure 16.5](#): Simple Event Lab with a listener class
- [Figure 16.6](#): Simple Event Lab with a listener object
- [Figure 16.7](#): Simple Event Lab continued
- [Figure 16.8](#): One listener object for many buttons
- [Figure 16.9](#): Simple Nim GUI
- [Figure 16.10](#): Nim Lab
- [Figure 16.11](#): Nim, with output to a text area
- [Figure 16.12](#): Nim with graphical output
- [Figure 16.13](#): Nim with graphical output, game in progress
- [Figure 16.14](#): Enabled and disabled buttons
- [Figure 16.15](#): Nim with disabled buttons
- [Figure 16.16](#): Check box and choice
- [Figure 16.17](#): Receiving events from a check box and a choice
- [Figure 16.18](#): Event Lab
- [Figure 16.19](#): Scrollbar and text field

Chapter 17: Final Project

- [Figure 17.1](#): Final Project
- [Figure 17.2](#): Final Project, with lines
- [Figure 17.3](#): Menu schematic
- [Figure 17.4](#): Teting the menu's look
- [Figure 17.5](#): Window, Frame, and FileDialog
- [Figure 17.6](#): File dialog box configured for opening
- [Figure 17.7](#): Too many radio buttons
- [Figure 17.8](#): Testing color selection
- [Figure 17.9](#): GUI layout
- [Figure 17.10](#): Positioning text

Appendix A: Downloading and Installing Java

- [Figure A.1](#): Windows SDK file layout

List of Tables

Chapter 1: An Introduction to Computers That Will Actually Help You in Life

[Table 1.1:](#) Opcodes

Chapter 2: Data

[Table 2.1:](#) Java's Integer Data Types

[Table 2.2:](#) Java's Floating-Point Data Types

[Table 2.3:](#) Java's Primitive Data Types

[Table 2.3:](#) Naming Consistency

Chapter 3: Operations

[Table 3.1:](#) Binary Bitwise Operations

[Table 3.2:](#) Comparison Operators

[Table 3.3:](#) Compound Assignment

[Table 3.4:](#) Ranges of Numeric Types

[Table 3.5:](#) Binary Arithmetic Result Types

[Table 3.6:](#) Operator Precedence

Chapter 8: Inheritance

[Table 8.1:](#) References, Variables, and Methods

Chapter 9: Packages and Access

[Table 9.1:](#) Legal Access Modes for Overriding Methods

Chapter 12: The Core Java Packages and Classes

[Table 12.1:](#) String Concatenation Conversion Rules

[Table 12.2:](#) Wrapper Class Names

Chapter 13: File Input and Output

[Table 13.1:](#) Byte -1 vs. Int -1

Chapter 14: Painting

[Table 14.1:](#) Combining Additive Primary Colors

Team LIB

Team LIB



Ground-Up Java

by Philip Heller

ISBN:0782141900

Sybex © 2003 (488 pages)

In addition to learning the core Java language, you will also acquire a broad understanding of vital programming concepts, including variables, control, memory, indirection, compilation, and calling.

Table of Contents

[Ground-Up Java](#)

[Introduction](#)

[Chapter 1](#) - An Introduction to Computers That Will Actually Help You in Life

[Chapter 2](#) - Data

[Chapter 3](#) - Operations

[Chapter 4](#) - Methods

[Chapter 5](#) - Conditionals and Loops

[Chapter 6](#) - Arrays

[Chapter 7](#) - Introduction to Objects

[Chapter 8](#) - Inheritance

[Chapter 9](#) - Packages and Access

[Chapter 10](#) - Interfaces

[Chapter 11](#) - Exceptions

[Chapter 12](#) - The Core Java Packages and Classes

[Chapter 13](#) - File Input and Output

[Chapter 14](#) - Painting

[Chapter 15](#) - Components

[Chapter 16](#) - Events

[Chapter 17](#) - Final Project

[Appendix A](#) - Downloading and Installing Java

[Appendix B](#) - Solutions to the Exercises

[Glossary](#)

[Index](#)

[List of Figures](#)

[List of Tables](#)

Team LIB

Team LIB